DISCRETE MATHEMATICAL PROOF WITH JUNIT

Guillermo Choque Aspiazu
San Andres University, La Paz-Bolivia
gchoquea@gmail.com

Abstract
In this paper we can develop a view of mathematical proofs in terms of his application to testing computational programs, we review the basic odd and even theory, the mathematical definitions of odd and even numbers, an initial weird proof of odd numbers, the unit testing for software programs, the JUnit testing framework, and finally we develop an discrete mathematical proof with Junit.
Keywords: *Mathematical proof, odd number, unit testing, Junit testing framework.*

1. INTRODUCTION

A mathematical proof is an argument which convinces other people that something is true. Math isn't a court of law, so a "preponderance of the evidence" or "beyond any reasonable doubt" isn't good enough. In principle we try to prove things beyond any doubt at all, although in real life people make mistakes, and total rigor can be impractical for large projects. Anyway, there is a certain vocabulary and grammar that underlies all mathematical proofs. The vocabulary includes logical words such as "or", "if", etc. These words have very precise meanings in mathematics which can differ slightly from everyday usage. By "grammar", We mean that there are certain common-sense principles of logic, or proof techniques, which you can use to start with statements which you know and deduce statements which you didn't know before (Hutchings, 2002)

In this brief paper we can present a view of mathematical proofs in terms of his application to testing computational programs, we review the basic odd and even theory, the mathematical definitions of odd and even numbers, an initial weird proof of odd numbers, the unit testing for software programs, the JUnit testing framework, and finally we develop an discrete mathematical proof with Junit.

2. ODD AND EVEN THEORY

Carasco (2010) says that, even and odd numbers are straightforward concepts. I will start easy, but I will try to challenge the topic a little bit. An even number is any number that can be divided by 2. For example, 12 can be divided by 2, so 12 is even. We saw in divisibility rules that a number is divisible by 2 if its last digit is 0,2,4,6,or 8. Therefore, any number whose last digit is 0, 2, 4, 6, or 8 is an even number. In a formal terms, a number n is even if there exist a number k, such that $n = 2k$ where k is an integer. This is formal way of saying that if n is divided by 2, we always get a quotient k with no remainder. Having no remainder means that n can in fact be divided by 2.

In the other side, an odd number is any number that cannot be divided by 2.For example, 25 cannot be divided by 2, so 25 is odd. We saw in divisibility rules that a number is divisible by 2 if its last digit is 0,2,4,6,or 8. Therefore, any number whose last digit is not 0, 2, 4, 6, or 8 is an odd number. Similarly to even numbers, the formal definition of an odd number says: A number n is odd if there exist a number k, such that $n=2k+1$ where k is an integer. This is formal way of saying that if n is divided by 2, we always get a quotient k with a remainder of 1. Having a remainder of 1 means that n cannot in fact be divided by 2.

The basic operations with even and odd numbers are:

- Addition:
  - even + even = even
  - even + odd = odd
  - odd + odd = even
- Multiplication
  - even × even = even
  - even × odd = even
  - odd × odd = odd
- Subtraction
  - even − even = even
  - even − odd = odd
  - odd − odd = even

Symbolically we can write, if $n \in Z$ then

- n is even $\Leftrightarrow \exists k \in Z$ such that $n = 2k$.
- n is odd $\Leftrightarrow \exists k \in Z$ such that $n = 2k + 1$.

3. A WEIRD PROOF

Contemplate the following:

$1 = 1$
$1+3 = 4$
$1+3+5 = 9$
$1+3+5+7 = 16$
$1+3+5+7+9 = 25$

It looks like the sum of the first n odd integers is n^2. Is it true? Certainly we cannot draw that conclusion from just the few above examples. But let us attempt to prove it. The nth odd number is $2n-1$, so our sum is the following:

$$\sum_{i=1}^{n}(2i-1)=1+3+5+...+(2n-1)=[1+3+5+...+(2n-3)]+(2n-1)$$

But if the sum of the first n odd integers is n^2 , then the sum of the first odd $n-1$ integers must be $(n-1)^2$. Therefore, our sum becomes:

$$(n-1)^2 + (2n-1) = n^2 - 2n + 1 + 2n - 1 = n^2$$

But is that legitimate? It seems that in our proof we used the very same fact that we are trying to prove! Or did we? In fact we did not. We used the fact for a smaller number ($n-1$ instead of n). Essentially, what we have established is the following: if the sum of the first $n-1$ integers is $(n-1)^2$, then the sum of the first n integers is n^2 . And this works for any n. All we need now is a base case for some value of n, say n_0 . But we have a base case because we enumerated few cases above. This technique is known as proof by induction. It is very simple, and least insightful. In deed, we did not gain any intuition why the sum of the first n odd integers is n^2 although we proved it (Saad, 2007).

Another heuristic way is doing the following: Let n be a natural number, it must be greater than 1. By definition

$$S = 1 + 3 + 5 + 7 + ... + (2n-3) + (2n-1)$$
$$S = (2n-1) + (2n-3) + ... + 5 + 3 + 1$$

Adding both terms

$$2S = 2n + 2n + ... + 2n + 2n \text{ (n times)}$$
$$2S = n*2n$$
$$S = n*n = n^2$$

4. UNIT TESTING

Unit testing have been around since the 1970's and was first introduced by Kent Beck in the Smalltalk language. Today, the concept has been adapted to a myriad of other languages. The definition of a unit test is as follows (Osherove, 2009): A unit test is a piece of a code, usually a method, that invokes another piece of code and checks the correctness of some assumptions afterward. If the assumptions turn out to be wrong, the unit test has failed. A "unit" is a method or function. There are also several properties a unit test must adhere to (Osherove, 2009):

- It should be automated and repeatable.
- It should be easy to implement.
- Once it's written, it should remain for future use.
- Anyone should be able to run it.
- It should run at the push of a button.
- It should run quickly.

With a unit test, the system under test (SUT) would be very small and perhaps only relevant to developers working closely with the code[5]. A unit test should only exercise; logical code, code that contains branches, calculations or in other ways enforces some decision-making. Simple property *getters* and *setters* in Java are examples of non-logical code.

Unit testing frameworks help developers write, run and review unit tests. These frameworks are commonly named xUnit frameworks[1] and share a set of features across implementations (Beck, 2003; Osherove, 2009):

- An assertion mechanism to decide the outcome of a test, usually an variant of *assertEquals(expected, actual)*.
- Fixtures for common objects or external resources among tests.
- A way of testing for exceptions.
- A representation of individual tests and test cases[2] to help structure test assets.
- An executable to run and review single tests or groups of tests.

1 Where the x represents the language it is created to test.
2 Test cases, also called test suites, are a way of representing a collection of tests.

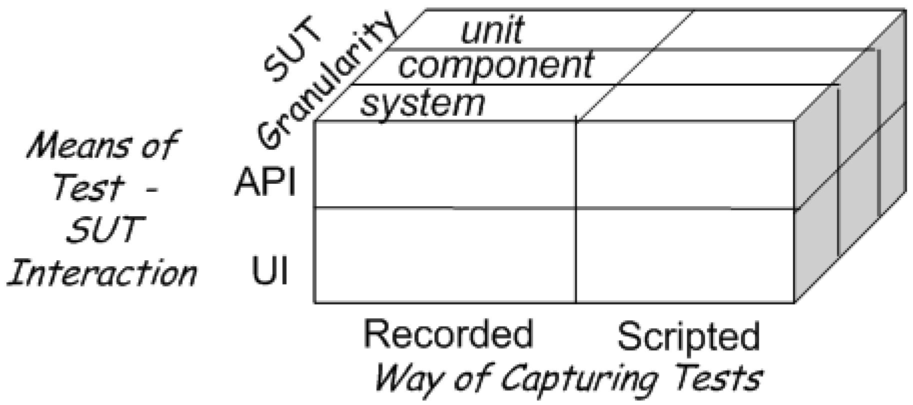


Figure 1: xUnit framework automation strategy and SUT granularity.
Source: Adapted from (Meszaros, 2007).

Figure 1 shows the automation strategy of the xUnit frameworks. Tests are hand-scripted on through an Application Programming Interface (API) on a unit level. This gives more robust and maintainable tests that even can be prebuilt. The tools required are simple and cheap, but require more skill to apply and an existing API. Looking at the front of Figure 1 other types of automation are given. Both scripted and recorded UI tests are available with the Selenium tool (Selenium, 2011). Also *FuncUnit* is in this category. These tools work primarily on a system granularity level. The recorded API testing exercises the SUT and logs interesting interaction points, and compares these with earlier runs or expected results. Currently, not many tool use this strategy. Recorded UI tests are performed by robot users and are currently very unstable and give small returns (Kleivane, 2011).

5. JUNIT TESTING FRAMEWORK

Although the importance of testing in developing correct programs has always been recognized, proponents of agile methods, among which eXtreme Programming (XP) is the most well-known approach, have recently emphasized the beneficial role of unit testing and test automation (Beck, 2000; Martin, 2003). For instance, XP proponents advocate the use of a test-first approach, i.e., the discipline of writing automated test cases before writing the code, summarized in the following motto: “Never write a line of functional code without a broken test case” (Beck, 2001).

Test automation, which allows for the automatic execution and verification of large number of test cases, is neither new nor specific to agile methods [8]. What is new to XP and agile methods is the tight integration of test automation with a test-first approach to code development. This allows testing to be done both early and often. Thus, frequent regression testing can be performed, ensuring that the software works correctly whenever changes are made, whether these changes result from the addition of new code to handle additional features, the modification of existing code to fix bugs, or from refactoring done on the existing code to improve its quality and maintainability (Fowler, 1999; Beck, 2003).

For such an approach to software construction to be feasible, appropriate tools for automating the testing process must be available. One well-known such tool is JUnit (Beck and Gamma, 1998), a unit testing framework for Java, which we briefly explain, with an example of discrete mathematics, in the following paragraph.

6. DISCRETE MATHEMATICAL PROOF WITH JUNIT

Suppose the problem presented in the paragraph 3, the problem is solved using the construction of two programs in the Java programming language, the first from a classical point of view and the second from a weird proof perspective as presented in the above paragraph.

The program was written in Java using the two points of view.

```
/**
 * @author Guillermo Choque Aspiazu
 */
import java.io.*;
public class AddOdd1 {
   public static void main (String[] args)  throws IOException
   {
      BufferedReader stdin =
            new BufferedReader(new InputStreamReader(System.in), 1);

      System.out.println("Please enter a number (>1)");
      String s1=stdin.readLine();
      int n = Integer.parseInt(s1);

      System.out.println("The odd sum of " + n + " numbers is " + Method1(n));
      System.out.println("The odd sum of " + n + " numbers is " + Method2(n));
   }
   /** Sum n odd numbers in traditional way */
   static int Method1(int n){
      int suma=0;
      for (int i=1; i<=n; i++)
         suma = suma + (i*2-1);
      return suma;
   }
   /** Sum n odd numbers in heuristic way */
    static int Method2(int n){
      return n*n;
   }
}
```

Unit tests ensure that code works as intended. They are also very helpful to ensure that the code still works as intended in case you need to modify code for fixing a bug or extending functionality. Having a high test coverage of your code allows you to continue developing features without having to perform lots of manual tests. After verifying the functionality of the program, the unit tests are run using JUnit

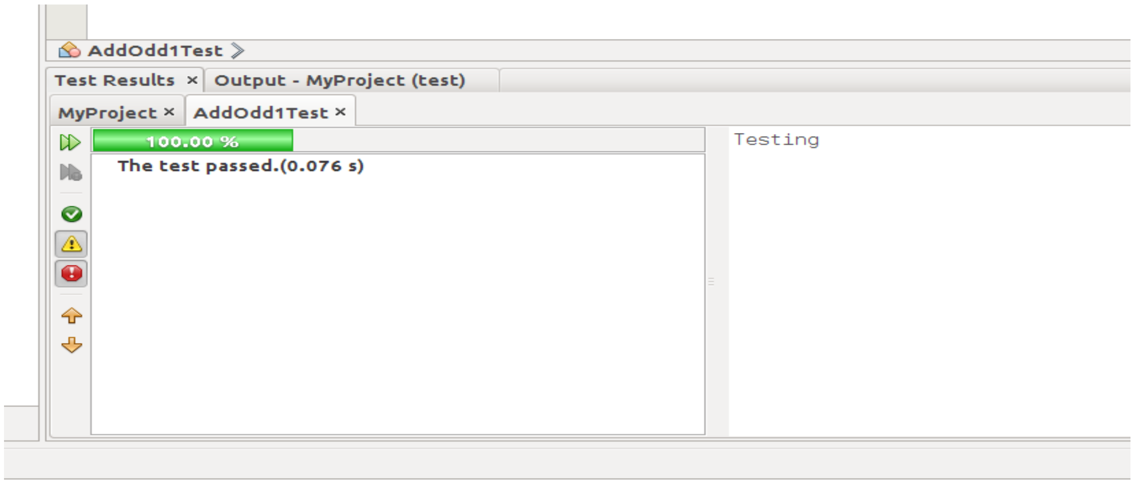
```
/**
 * @author Guillermo Choque Aspiazu
 */
import org.junit.AfterClass;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;

public class AddOdd1Test {
    AddOdd1 t;
    public AddOdd1Test() {
       t = new AddOdd1 ();
    }

    @Test
    public void testMain() throws Exception {
        System.out.println("Testing");
        assertEquals(t.Method1(5), 25);
        assertEquals(t.Method1(5), t.Method2(5));
    }
}
```

The tests were performed using the right and wrong cases. The example used to execute the rigth way was the sum of the first five odd natural number, the result is the number 25 which is equal to 5^2. In the wrong way we use the first four odd numbers whose result should be 16 and 25 was used.

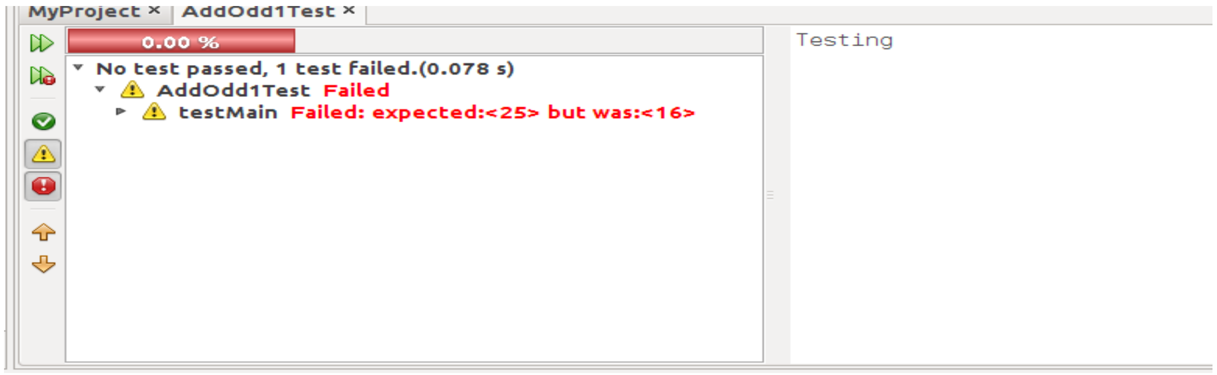
```
  @Test
    public void testMain() throws Exception {
        System.out.println("Testing");
        assertEquals(t.Method1(5), 25);
        assertEquals(t.Method1(5), t.Method2(5));
    }
```



```
The test passed.(0.076 s)
```

```
Testing
```

```
@Test
    public void testMain() throws Exception {
        System.out.println("Testing");
        assertEquals(t.Metodo1(5), 25);
        assertEquals(t.Metodo1(5), t.Metodo2(4));
    }
```



```
No test passed, 1 test failed.(0.078 s)
  AddOdd1Test  Failed
    testMain  Failed: expected:<25> but was:<16>
```

7. CONCLUSIONS

In this brief paper, different basic concepts of test used in discrete mathematics were developed; also we showed a brief conceptual framework about odd and even numbers, also the way in which you can use to generate the sum of the first odd natural numbers using heuristic means. Concepts of testing programs used in the Java programming language is also presented and how they can be used to test source code using JUnit framework.

8. ACKNOWLEDGEMENTS

This work was developed in the informatics laboratory of Umeå University using NetBeans IDE 7.3, I would like to thank Dr. Carl-Johan Orre for the ride from the high mountains of Bolivia and also for providing the laboratory of informatics. I would like to thank Dr. Göran Landgren by precise corrections to this article and finally I would like to thank MSc. Jorge Teran for their corrections to the Java program.

REFERENCES

- Beck, K. (2000) Extreme Programming Explained — Embrace Change. Addison-Wesley, Reading, MA.
- Beck, K. (2001) Aim, fire. IEEE Software, 18(6):87–89.
- Beck, K. (2003) Test-Driven Development By Example. Addison-Wesley.
- Beck, K. (2003) Test-Driven Development: By Example. Addison-Wesley.
- Beck, K. and Gamma, E. (1998) Test infected: Programmers love writing tests. Java Report, 3(7):37–50.
- Carasco, Jetser (2010) Even and odd numbers. Basic-mathematics.com. Recovered from: http://www.basic-mathematics.com/even-and-odd-numbers.html [Cited Jan 2014]
- Fewster, M. (1999) Software Test Automation. Addison-Wesley.
- Fowler, M. (1999) Refactoring — Improving the Design of Existing Code. Addison-Wesley, Reading, MA, 1999.
- Hutchings, Michael (2002) Introduction to mathematical arguments. Background handout for courses requiring proofs. Recovered from: http://math.berkeley.edu/~hutching/teach/proofs.pdf

[Cited Jan 2014]

- Kleivane, Tine Flåten (2011) Unit Testing with TDD in JavaScript. Report of master thesis. Master of Science in Computer Science. Norwegian University of Science and Technology. Department of Computer and Information Science. Recovered from: http://www.diva-portal.org/smash/get/diva2:450286/FULLTEXT01.pdf [Cited Jan 2014]
- Martin, R.C. (2003) Agile Software Development — Principles, Patterns, and Practices. Prentice-Hall, Upper Saddle River, NJ.
- Meszaros, G. (2007) xUnit Test Patterns: Refactoring Test Code. Addison-Wesley.
- Mneimneh, Saad (2007) Discrete Mathematics. Inductive proofs. Lecture notes, Recovered from: http://www.cs.hunter.cuny.edu/~saad/courses/dm/notes/note5.pdf [Cited Jan 2014]
- Osherove, R. (2009) The Art of Unit Testing: with Examples in .NET. Manning Publications.
- Selenium (2011) Platforms Supported by Selenium. Recovered from: http://seleniumhq.org/about/platforms.html#programming-languages [Cited May 2011].