http://securesoftwaredev.com/2011/12/05/practicing-tdd-using-the-roman-numerals-kata/

# Primer Test

[TestMethod]  
public void RetornaICuandoElDecimalEs1()  
{  
    var roman = RomanNumerals.ToRoman(1);//ArabicToRoman o Convert  
    Assert.AreEqual(roman, "I");  
}

Este paso no es solo sobre testing, sino sobre diseñar nuestra API desde la perspectiva.

public static string ToRoman(int i)  
{  
    return "I";  
}

# 2do Test

[TestMethod]  
public void RetornaIICuandoElDecimalEs2()  
{  
    var roman = RomanNumerals.ToRoman(2);  
    Assert.AreEqual(roman, "II");  
}

public static string ToRoman(int i)  
{  
    if (i==2)  
    {  
        return "II";  
    }  
    return "I";  
}

OK, we’re Green. Now let’s look at this code. It’s pretty obvious that if we continue down this path, we’ll end up with very, very bad code. There is no design at all, just a bunch of hacks. That’s why the Refactor step is essential.

We pass the tests, but how about duplication? There is duplication in the Arabic number passed in and the number of I’s the method returns. This may not be obvious to everyone because of the returnin the middle of the method. Let’s get rid of it to better expose the duplication…

# Refactor

public static string ToRoman(int number)  
{  
    var result = new StringBuilder();  
    for (int i = 0; i < number; i++)  
    {  
        result.append("I");  
    }  
    return result.ToString();  
}

What we did here was generalize an if statement into a for (orwhile). This is one of a bunch of transformations one often uses in TDD. The net effect of a generalization like this is that we have discovered a rule based on similarities. This means that our code can now handle more cases than the ones we supplied as tests.

# 3er Test

[TestMethod]  
public void RetornaIIICuandoElDecimalEs3()  
{  
    var roman = RomanNumerals.ToRoman(3);  
    Assert.AreEqual(roman, "III");  
}

# 4to Test

[TestMethod]  
public void RetornaIVCuandoElDecimalEs4()  
{  
    var roman = RomanNumerals.ToRoman(4);  
    Assert.AreEqual(roman, "IV");  
}

public static string ToRoman(int number)  
{  
    var result = "";  
    if (number == 4)  
    {  
        result = "IV";  
    }  
    else  
    {  
        for (int i = 0; i < number; i++)  
        {  
            result += "I";  
        }  
    }  
    return result;  
}

# 5to Test

[TestMethod]  
public void RetornaVCuandoElDecimalEs5()  
{  
    var roman = RomanNumerals.ToRoman(5);  
    Assert.AreEqual(roman, "V");  
}

public static string ToRoman(int number)  
{  
    var result = "";  
    if (number==5)  
    {  
        result = "V";  
    }else if (number == 4)  
    {  
        result = "IV";  
    }  
    else  
    {  
        for (int i = 0; i < number; i++)  
        {  
            result += "I";  
        }  
    }  
    return result;  
}

This is turning into a mess. But there is no duplication apparent yet, and the code does sort of say what we mean. So let’s push our uneasy feelings aside for a little while and move on to the next test:

# 6to Test

[TestMethod]  
public void RetornaVICuandoElDecimalEs6()  
{  
    var roman = RomanNumerals.ToRoman(6);  
    Assert.AreEqual(roman, "VI");  
}

public static string ToRoman(int number)  
{  
    var result = "";  
    if (number == 6)  
    {  
        result = "VI";  
    }  
    else if (number == 5)  
    {  
        result = "V";  
    }  
    else if (number == 4)  
    {  
        result = "IV";  
    }  
    else  
    {  
        for (int i = 0; i < number; i++)  
        {  
            result += "I";  
        }  
    }  
    return result;  
}

Hmm, uglier still, but at least some duplication is now becoming visible: VI is V followed by I, and we already have code to append those. So we could first add the V and then rely on the for loop to add the I:

# Refactor

public static string ToRoman(int number)  
{  
    var result = "";  
    if (number >= 5)  
    {  
        result = "V";  
        number -= 5;  
    }  
    if (number == 4)  
    {  
        result = "IV";  
        number -= 4;  
    }  
    for (int i = 0; i < number; i++)  
    {  
        result += "I";  
    }  
    return result;  
}

# Deberíamos probar el 7 y el 8?

# 9no Test

[TestMethod]  
public void RetornaIXCuandoElDecimalEs9()  
{  
    var roman = RomanNumerals.ToRoman(9);  
    Assert.AreEqual(roman, "IX");  
}

public static string ToRoman(int number)  
{  
    var result = "";  
    if (number==9)  
    {  
        result = "IX";  
        number -= 9;  
    }  
    if (number >= 5)  
    {  
        result = "V";  
        number -= 5;  
    }  
    if (number == 4)  
    {  
        result = "IV";  
        number -= 4;  
    }  
    for (int i = 0; i < number; i++)  
    {  
        result += "I";  
    }  
    return result;  
}

There’s definitely a pattern emerging. Two ifs use ==, and one uses>=, but the rest of the statements is the same. We can make them all completely identical by a slight generalization:

# Refactoring (Emerge un Patrón, generalizamos los condicionales)

public static string ToRoman(int number)  
{  
    var result = "";  
    if (number>=9)  
    {  
        result = "IX";  
        number -= 9;  
    }  
    if (number >= 5)  
    {  
        result = "V";  
        number -= 5;  
    }  
    if (number >= 4)  
    {  
        result = "IV";  
        number -= 4;  
    }  
    for (int i = 0; i < number; i++)  
    {  
        result += "I";  
    }  
    return result;  
}

# Extraer la duplicación

public static string ToRoman(int number)  
{  
    var result = new StringBuilder();  
    var remaining = number;  
    remaining = AppendRomanNumerals(remaining, 9, "IX", result);  
    remaining = AppendRomanNumerals (remaining, 5, "V", result);  
    remaining = AppendRomanNumerals (remaining, 4, "IV", result);  
    for (int i = 0; i < remaining; i++)  
    {  
        result.Append("I");  
    }  
    return result.ToString();  
}  
  
private static int AppendRomanNumerals (int remaining, int arabic, string roman, StringBuilder result)  
{  
    if (remaining >= arabic)  
    {  
        result.Append(roman);  
        remaining -= arabic;  
    }  
    return remaining;  
}

Aún hay duplicación

private static int[] VALUES = { 9, 5, 4 };  
private static string[] SYMBOLS = { "IX", "V", "IV" };  
public static string ToRoman(int number)  
{  
    var result = new StringBuilder();  
    var remaining = number;  
    for (int i = 0; i < VALUES.Length; i++)  
    {  
        remaining = AppendRomanNumerals(remaining, VALUES[i], SYMBOLS[i], result);  
    }  
    for (int i = 0; i < remaining; i++)  
    {  
        result.Append("I");  
    }  
    return result.ToString();  
}

Now that we look at the code this way, it seems that the for loop does something similar to what appendRomanNumerals does. The only difference is that the loop does it multiple times, while the method does it only once. We can generalize the method and rewrite the loop to make this duplication more visible:

 public static string ToRoman(int number)  
 {  
     var result = new StringBuilder();  
     var remaining = number;  
     for (int i = 0; i < VALUES.Length; i++)  
     {  
         remaining = AppendRomanNumerals(remaining, VALUES[i], SYMBOLS[i], result);  
     }  
     while (remaining >= 1)  
     {  
         result.Append("I");  
         remaining -= 1;  
     }  
     return result.ToString();  
 }  
  
 private static int AppendRomanNumerals(int remaining, int arabic, string roman, StringBuilder result)  
 {  
     while(remaining >= arabic)  
     {  
         result.Append(roman);  
         remaining -= arabic;  
     }  
     return remaining;  
 }

Eliminamos duplicados

private static int[] VALUES = { 9, 5, 4,1 };  
private static string[] SYMBOLS = { "IX", "V", "IV","I" };  
public static string ToRoman(int number)  
{  
    var result = new StringBuilder();  
    var remaining = number;  
    for (int i = 0; i < VALUES.Length; i++)  
    {  
        remaining = AppendRomanNumerals(remaining, VALUES[i], SYMBOLS[i], result);  
    }  
    return result.ToString();  
}

## Ejemplos Grandes

            Assert.AreEqual(RomanNumerals.ToRoman(14), "XIV");

            Assert.AreEqual(RomanNumerals.ToRoman(20), "XX");

            Assert.AreEqual(RomanNumerals.ToRoman(40), "XL");

            Assert.AreEqual(RomanNumerals.ToRoman(50), "L");  
            Assert.AreEqual(RomanNumerals.ToRoman(395), "CCCXCV");

            Assert.AreEqual(RomanNumerals.ToRoman(2499), "MMCDXCIX");  
            Assert.AreEqual(RomanNumerals.ToRoman(3949), "MMMCMXLIX");

static int[] ARABIC\_DIGITS =

{ 1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1 };

static String[] ROMAN\_DIGITS =

{ "M","CM","D","CD","C","XC","L","XL","X","IX","V","IV","I" };

# ALTERNATIVA

public class RomanNumerals  
{  
    static int[] ARABIC\_DIGITS = {10, 9, 5, 4 };  
    static string[] ROMAN\_DIGITS = {"X", "IX", "V", "IV" };  
    public static string ToRoman(int arabic)  
    {  
        var result = "";  
  
        for (int i = 0; i < ARABIC\_DIGITS.Length; i++)  
        {  
            if (arabic >= ARABIC\_DIGITS[i])  
            {  
                result += ROMAN\_DIGITS[i];  
                arabic -= ARABIC\_DIGITS[i];  
            }  
        }  
  
        while (arabic >= 1)  
        {  
            result += "I";  
            arabic--;  
        }  
  
        return result;  
    }  
  
}