TDD Helping Algorithm Development

Contents

[Benefits of Testing 2](#_Toc53414226)

[Background 2](#_Toc53414227)

[The Roman Numerals TDD Demo 3](#_Toc53414228)

[Create a Project 3](#_Toc53414229)

[Algorithm Development 6](#_Toc53414230)

[Simple Additive Notation 6](#_Toc53414231)

[Subtractive Notation 8](#_Toc53414232)

[Another Implementation Approach 11](#_Toc53414233)

[Performance 13](#_Toc53414234)

[Resources 13](#_Toc53414235)

[References 13](#_Toc53414236)

Extreme Programming (**XP**) introduced automated testing as a first-class part of software development in 1999, and it used testing both in development and maintenance. Tests help during development, later with enhancement, and finally in understanding the code. This article shows an example of applying testing practices during the initial construction phase of an algorithm. A previous companion article describes how a complete body of regression tests aids modifications for performance (see Test Driven Development Really Works in resources #2 below.) *Please see my GitHub repository that hosts code this article* (see reference #3.) Now let’s see how testing helps during the construction phase, and is called Test Driven Development (**TDD**.)

# Benefits of Testing

Testing enhances building functionality through feature isolation and incremental delivery, preserves application stability through regression, and facilitates debugging. TDD offers "observability” into the operation of application components so the components can be tested and modified without an entire application “context” and deployment during development cycles.

# Background

I attended an excellent lecture on Test Drive Development (TDD) given by Mark Shead (see resources #1), in which he used a “Arabic to Roman Numeral” conversion algorithm to demonstrate how TDD enabled exploring code and algorithm options. He of course started with the TDD mantra of:

* Write a failing test first.
* Write code that makes the test succeed.
* Repeat.

An important goal of this approach is to keep the software very close to working all the time. This highly incremental approach works well when the algorithm being implemented is straight-forward and does not require significant abstraction or research. Given the *purpose* of Mark’s lecture, which was to illustrate TDD in action, this orthodox approach was reasonable.

Often however, algorithms will require significant abstraction and research during their implementation. For example, suppose we are defining the notion of distance between two “points” ***a*** and ***b***. We might start with a pair of one-dimensional points and compute the distance between them as:

* D = ABS(***A***x – ***B***x)

We easily write the appropriate tests for one dimension. Next we solve the problem in two dimensions (the X-Y plane), using Euclidian distance:

* D = SQRT((***A***x – ***B***x)^2 + (***A***y – ***B***y)^2)

Again, we easily write our tests and proceed to three dimensions. Soon we realize that there is a general formula for the distance between two points (***a*** and ***b***), for ***N*** *> 0* dimensions:

* D = SQRT( SUM( (***a***[i] – ***b***[i])^2, i=1..***N*** ) )

Now we have to refactor both our solution and our tests to take this more general approach. A little research in advance, revealing the potential abstraction simplifying the problem, would have enabled us to write less code, and require much less refactoring to take the algorithm to the next implementation stage.

The lessons learned from this initial example are:

1. You may need some research into your algorithm prior to writing any code at all.
2. Refine concepts and representations prior to writing code for tests and the implementation.
3. Bear in mind that algorithm exploration will often result in refactoring tests.

# The Roman Numerals TDD Demo

Borrowing from Mark’s presentation, we are developing a *Roman Numeral Conversion* utility and concentrating on converting dates, represented as binary integers, to a string representation using Roman Numerals. We review the requirements and proposed solution during our design review. From the review, we note that:

1. The Wiki reference gives a good overview of Roman Numerals (reference #1.)
2. Based on reviewing above, the conversion problem is related to the change-making problem (reference #2.)
3. Our colleagues noted that this problem is also similar to radix conversion, as non-unity Roman Numerals are all multiples of five.

Our application of Roman Numerals is primary intended for dates, and we place an upper limit of 3,999 that we will convert. Negative integers are disallowed. Finally, we noted a nuance in representing numbers with Roman Numerals: there is an *additive* notation and a *subtractive* notation. For example, the number four has two potential representations: “IIII” (additive) and “IV” (subtractive). In general, the subtractive representation is preferred, but traditional clock representations sometimes improperly use the additive form only.

## Create a Project

We begin by creating a Maven-based eclipse project. We layout the structure and initial classes in the project:

| pom.xml

|

+---src

| +---main

| | +---java

| | | \---don

| | | \---demodev

| | | \---romannumerals

| | | Converter.java (Interface defining converter API)

| | | ConverterImpl.java (Initial Implementation)

| | \---resources

| \---test

| +---java

| | \---don

| | \---demodev

| | \---romannumerals

| | \---test

| | Arabic2RomanBaseTest.java (Initial tests, with first failing test)

| |

| \---resources

We initially entered a little more code than the orthodox TDD approach would dictate, but we will get good value from that initial entry. Defining the interface first allows us to control how users will see the converter, and it allows us to later modify the implementation in many ways by separating API from implementation.

Our Converter interface holds definitions of the problem that defines aspects of the solution algorithm:

**public** **static** **final** **class** Roman2Arabic {

**public** **final** String romanSymbol;

**public** **final** **int** arabic;

**public** Roman2Arabic(String romanSymbol, **int** arabic) {

**super**();

**this**.romanSymbol = romanSymbol;

**this**.arabic = arabic;

}

@Override

**public** String toString() {

**return** "Roman2Arabic [Symbol=" + romanSymbol + ", Arabic=" + arabic + "]";

}

}

/\*\*

\* Table of Roman numerals and associated integer value, ordered from largest

\* value numeral to smallest value.

\*/

**public** **static** **final** Roman2Arabic[] ***mapping*** = { **new** Roman2Arabic("M", 1000), **new** Roman2Arabic("D", 500),

**new** Roman2Arabic("C", 100), **new** Roman2Arabic("L", 50), **new** Roman2Arabic("X", 10),

**new** Roman2Arabic("V", 5), **new** Roman2Arabic("I", 1) };

We begin testing by checking preconditions and validations done by the initial implementation. We do this before beginning functionality development. Our first functional test is with a simple clock time representation, and traditionally it fails. Here is our initial test class:

**public** **class** Arabic2RomanBaseTest {

**protected** Converter converter;

@Before

**public** **void** setUp() **throws** Exception {

converter = **new** ConverterImpl();

}

@After

**public** **void** tearDown() **throws** Exception {

converter = **null**;

}

@Test

**public** **void** testZeroParameter() {

Assert.*assertEquals*("small param", "", converter.arabic2Roman(0));

}

// Validation test sequence

@Test(expected = IllegalArgumentException.**class**)

**public** **void** testNegativeParameterFails() {

Assert.*assertEquals*("negative param fails", "BAD", converter.arabic2Roman(-1));

}

@Test(expected = IllegalArgumentException.**class**)

**public** **void** testBigParameterFails() {

Assert.*assertEquals*("Big param fails", "BAD", converter.arabic2Roman(4001));

}

@Test

**public** **void** testInitial0() {

Assert.*assertEquals*("0 failed", "", converter.arabic2Roman(0));

}

// Functionality testing -- additive numerals

@Test

**public** **void** testClockAdditive() {

Assert.*assertEquals*("One failed", "I", converter.arabic2Roman(1));

}

}

# Algorithm Development

Reviewing the common greedy algorithm approach for making change (money), we note that the “largest coins” are used first, with remaining amounts of change likewise decreased using the next largest coin, until the entire change amount is accumulated. We apply this approach to our converter to implement the *additive* notation for Roman numerals. Once we have a basic Roman numeral conversion algorithm, we refine it to use the *subtractive* notation.

## Simple Additive Notation

Restating the first two of four rules from reference #4 defines the *additive* notation:

1. If one or more letters are placed after another letter of greater value, add that amount.

VI = 6 (5 + 1 = 6)

XXVII = 27 (10 + 10 + 5 + 1 + 1 = 27)

MDC = 1,600 (1,000 + 500 + 100 = 1,600)

1. A letter cannot be repeated more than three times.

30 = XXX (10 + 10 + 10 = 30)

40 = XL (50 - 10 = 40) You cannot write 40 as XXXX

We ignore the second rule for now, and the simple additive algorithm is:

**private** String convertRoman2ArabicImpl(**final** **int** arabic) {

**final** StringBuilder roman = **new** StringBuilder();

**int** numeralIndex = 0;

**int** value = arabic;

**while** (value > 0) {

**int** romanValue = Converter.***mapping***[numeralIndex].arabic;

**int** howMany = value / romanValue;

**if** (howMany > 0) {

**for** (**int** i = 0; i < howMany; i++) {

roman.append(Converter.***mapping***[numeralIndex].romanSymbol);

}

value -= howMany \* Converter.***mapping***[numeralIndex].arabic;

}

numeralIndex++;

}

**return** roman.toString();

}

The corresponding unit tests are:

@Test

**public** **void** testSingleRomanNumeral() {

**for** (Roman2Arabic ra : Converter.***mapping***) {

Assert.*assertEquals*("Mapping for " + ra.arabic + " failed", ra.romanSymbol,

converter.arabic2Roman(ra.arabic));

}

}

@Test

**public** **void** testClockAdditive() {

Assert.*assertEquals*("One failed", "I", converter.arabic2Roman(1));

Assert.*assertEquals*("two failed", "II", converter.arabic2Roman(2));

Assert.*assertEquals*("three failed", "III", converter.arabic2Roman(3));

// Assert.assertEquals("four failed", "IIII", converter.arabic2Roman(4)); fails

// because too many I symbols

Assert.*assertEquals*("four failed", "IV", converter.arabic2Roman(4));

Assert.*assertEquals*("five failed", "V", converter.arabic2Roman(5));

Assert.*assertEquals*("six failed", "VI", converter.arabic2Roman(6));

Assert.*assertEquals*("seven failed", "VII", converter.arabic2Roman(7));

Assert.*assertEquals*("eight failed", "VIII", converter.arabic2Roman(8));

// Assert.assertEquals("nine failed", "VIIII", converter.arabic2Roman(9)); fails

// because too many I symbols

Assert.*assertEquals*("nine failed", "IX", converter.arabic2Roman(9));

Assert.*assertEquals*("ten failed", "X", converter.arabic2Roman(10));

Assert.*assertEquals*("eleven failed", "XI", converter.arabic2Roman(11));

Assert.*assertEquals*("twelve failed", "XII", converter.arabic2Roman(12));

}

@Test

**public** **void** testGeneralAdditive() {

Assert.*assertEquals*("27 failed", "XXVII", converter.arabic2Roman(27));

Assert.*assertEquals*("27 failed", "XXVII", converter.arabic2Roman(27));

Assert.*assertEquals*("30 failed", "XXX", converter.arabic2Roman(30));

Assert.*assertEquals*("1600 failed", "MDC", converter.arabic2Roman(1600));

Assert.*assertEquals*("1161 failed", "MCLXI", converter.arabic2Roman(1161));

Assert.*assertEquals*("2020 failed", "MMXX", converter.arabic2Roman(2020));

**int** sum = Arrays.*stream*(Converter.***mapping***).map(n -> n.arabic).mapToInt(Integer::intValue).sum();

Assert.*assertEquals*("all digits sum value, " + sum + ", failed", "MDCLXVI", converter.arabic2Roman(sum));

}

## Subtractive Notation

We can further reduce the remaining Arabic value to be converted using *subtractive* notation. One can think of this leftover amount as representing the remainder of the Arabic number after the initial representation. Here are the final three rules from reference #4 outlining the subtractive notation:

1. If a letter is placed before another letter of greater value, subtract that amount.

IX = 9 (10 - 1 = 9)

XL = 40 (50 - 10 = 40)

CML = 950 (900 + 50 = 950)

1. You can only subtract powers of 10 (I, X, C).

95 = XCV (100 - 10 + 5 = 95)

You cannot write 95 as VC because V is not a power of 10.

1. You cannot subtract more than one number from another number.

18 = XVIII (10 + 5 + 1 + 1 + 1 = 18)

You cannot write 18 as IIXX.

We refactor our algorithm to allow us to develop the subtractive notation, and we now incorporate the rule #2 above limiting consecutive symbols to three. Two clock unit tests need to be commented out because the simple additive notation can’t use more than three consecutive symbols. Our initial refactoring, combining additive and subtractive notation, is:

**private** String convertRoman2ArabicImpl(**final** **int** arabic) {

**final** StringBuilder roman = **new** StringBuilder();

**int** numeralIndex = 0;

**int** value = arabic;

**while** (value > 0) {

RomanNotation romanNotation = useAdditiveNotation(value, numeralIndex);

**if** (romanNotation.subtracted > 0) {

value -= romanNotation.subtracted;

roman.append(romanNotation.romanNumerals);

}

**if** (value > 0) {

romanNotation = useSubtractiveNotation(value, numeralIndex);

**if** (romanNotation.subtracted > 0) {

value -= romanNotation.subtracted;

roman.append(romanNotation.romanNumerals);

}

}

numeralIndex++;

}

**return** roman.toString();

}

**public** RomanNotation useAdditiveNotation(**final** **int** arabic, **final** **int** numeralIndex) {

**if** (arabic < 1) {

**return** **new** RomanNotation("", 0);

}

**final** **int** romanValue = Converter.***mapping***[numeralIndex].arabic;

**final** **int** howMany = arabic / romanValue;

**if** (howMany < 1) {

**return** **new** RomanNotation("", 0);

}

**final** StringBuilder roman = **new** StringBuilder();

**final** **int** limitedUsed = Math.*min*(3, howMany);

**for** (**int** i = 0; i < limitedUsed; i++) {

roman.append(Converter.***mapping***[numeralIndex].romanSymbol);

}

**return** **new** RomanNotation(roman.toString(), limitedUsed \* romanValue);

}

**public** **static** **final** **int**[] ***powerOf10*** = { 2, 4, 6 }; // C, X, I

**public** RomanNotation useSubtractiveNotation(**final** **int** arabic, **final** **int** numeralIndex) {

**final** **int** romanValue = Converter.***mapping***[numeralIndex].arabic;

**if** (arabic < 1 || numeralIndex == Converter.***mapping***.length - 1) {

**return** **new** RomanNotation("", 0);

}

//

// Try to subtract the largest value possible

**for** (**int** i = ***powerOf10***.length - 1; i >= 0; i--) {

**int** subtracted = romanValue - ***mapping***[***powerOf10***[i]].arabic;

**if** (subtracted > 0 && subtracted <= arabic) {

**return** **new** RomanNotation(***mapping***[***powerOf10***[i]].romanSymbol + ***mapping***[numeralIndex].romanSymbol, subtracted);

}

}

**return** **new** RomanNotation("", 0);

}

We also have specific tests for the *subtractive* notation:

@Test

**public** **void** testSingleSubtractiveByPower() {

Assert.*assertEquals*("4 = 5 - 1", "IV", converter.arabic2Roman(4));

Assert.*assertEquals*("40 = 50 - 10", "XL", converter.arabic2Roman(40));

Assert.*assertEquals*("900 = 1000 - 100", "CM", converter.arabic2Roman(900));

}

@Test

**public** **void** testMultipleSubtractive() {

Assert.*assertEquals*("1909 failed", "MCMIX", converter.arabic2Roman(1909));

Assert.*assertEquals*("1949 failed", "MCMIL", converter.arabic2Roman(1949));

}

@Test

**public** **void** testGeneralSubtractive() {

Assert.*assertEquals*("9 failed", "IX", converter.arabic2Roman(9));

Assert.*assertEquals*("49 failed", "IL", converter.arabic2Roman(49));

Assert.*assertEquals*("95 failed", "XCV", converter.arabic2Roman(95));

Assert.*assertEquals*("LARGEST value 3999, failed", "MMMIM", converter.arabic2Roman(3999));

}

# Another Implementation Approach

I can just hear Mark complaining that the iterative solution we have developed is not a “functional” solution, and that he would like to see a functional implementation. We can easily create a new functional implementation and reuse our tests to validate that it is correct. Here is a recursive solution:

**private** RomanNumeralNotation convert2Roman(**final** RomanNumeralNotation input, **final** **int** numeralIndex) {

**if** (input.remainder < 1) {

**return** input;

}

**final** RomanNumeralNotation current = useSubtractiveNotation(useAdditiveNotation(input, numeralIndex),

numeralIndex);

**return** current.remainder == 0 ? current : convert2Roman(current, numeralIndex + 1);

}

**public** RomanNumeralNotation useAdditiveNotation(**final** RomanNumeralNotation input, **final** **int** numeralIndex) {

**if** (input.remainder < 1) {

**return** input;

}

**final** **int** romanValue = Converter.***mapping***[numeralIndex].arabic;

**final** **int** howMany = input.remainder / romanValue;

**if** (howMany < 1) {

**return** input;

}

**final** StringBuilder roman = **new** StringBuilder(input.romanNumerals);

**final** **int** limitedUsed = Math.*min*(3, howMany);

**for** (**int** i = 0; i < limitedUsed; i++) {

roman.append(Converter.***mapping***[numeralIndex].romanSymbol);

}

**return** **new** RomanNumeralNotation(roman.toString(), input.remainder - limitedUsed \* romanValue);

}

**public** **static** **final** **int**[] ***powerOf10*** = { 2, 4, 6 }; // C, X, I

**public** RomanNumeralNotation useSubtractiveNotation(**final** RomanNumeralNotation input, **final** **int** numeralIndex) {

**final** **int** romanValue = Converter.***mapping***[numeralIndex].arabic;

**if** (input.remainder < 1 || numeralIndex == Converter.***mapping***.length - 1) {

**return** input;

}

//

// Try to subtract the largest value possible

**for** (**int** i = ***powerOf10***.length - 1; i >= 0; i--) {

**final int** subtracted = romanValue - ***mapping***[***powerOf10***[i]].arabic;

**if** (subtracted > 0 && subtracted <= input.remainder) {

**return** **new** RomanNumeralNotation(

input.romanNumerals + ***mapping***[***powerOf10***[i]].romanSymbol + ***mapping***[numeralIndex].romanSymbol, input.remainder - subtracted);

}

}

**return** input;

}

We leverage our previous iterative tests as follows:

**public** **class** Arabic2RomanRecursiveTest **extends** Arabic2RomanBaseTest {

@Override

@Before

**public** **void** setUp() **throws** Exception {

converter = **new** ConverterImplRecursive();

}

}

# Performance

While violating many unit-test dictums, it is fun to add a little performance comparison between our implementations to the unit test. Because of the stability offered by good regression test coverage, we are able to play around with optimizations. On my machine, we have:

* ConverterImpl required 11872.8 us for 10000 operations.
* ConverterImplRecursive required 18173.1 us for 10000 operations.

The recursive solution requires about 50% more computational effort. Good testing allows us to be courageous and try different solution approaches and improve our code.

# Resources

1. Mark Shead (<https://www.linkedin.com/in/markshead/>), president of Xeric Corporation (<http://www.xeric.net/>.)
2. My previous article on TDD: <https://www.linkedin.com/pulse/test-driven-development-tdd-really-works-donald-trummell-1c/>.

# References

1. Wikipedia discussion of Roman Numerals: <https://en.wikipedia.org/wiki/Roman_numerals>.
2. Wikipedia discussion of the Change-Making Problem: <https://en.wikipedia.org/wiki/Change-making_problem>.
3. My GitHub repository hosting this article: <https://github.com/DonaldET/DemoDev/tree/master/dev-topics-algorithms/dev-topics-romannumerals>.
4. Rules for writing roman numerals: <http://www.solano.edu/academic_success_center/forms/math/Roman%20Numerals.pdf>.