TDD And Algorithm Development

Contents

[Introduction 1](#_Toc49095639)

[Benefits of Testing 2](#_Toc49095640)

[Background 2](#_Toc49095641)

[The Roman Numerals Demo 3](#_Toc49095642)

[Create a Project 3](#_Toc49095643)

[Initial Tests 5](#_Toc49095644)

[Algorithm Development 6](#_Toc49095645)

[Simple Additive Notation 6](#_Toc49095646)

[Subtractive Notation 7](#_Toc49095647)

[Resources 8](#_Toc49095648)

[References 8](#_Toc49095649)

# Introduction

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

# 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 then the algorithm being implemented is straight-forward and does not require significant abstraction or research. Given the *purpose* of his 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”. We might start with a pair of one-dimensional points and compute the distance between them as:

* D = ABS(X1 – X2)

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((X1 – X2)^2 + (Y1 – Y2)^2)

Again we easily write our tests and proceed to three dimensions. Soon we realize that there is a general formula for ***N*** *> 0* dimensions.

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 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 Demo

Borrowing from Mark’s presentation, we are developing a Roman Numeral Conversion utility and concentrating on converting 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 4,000 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 use the additive form.

## Create a Project

We begin by creating a Maven-based eclipse project. We layout the 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

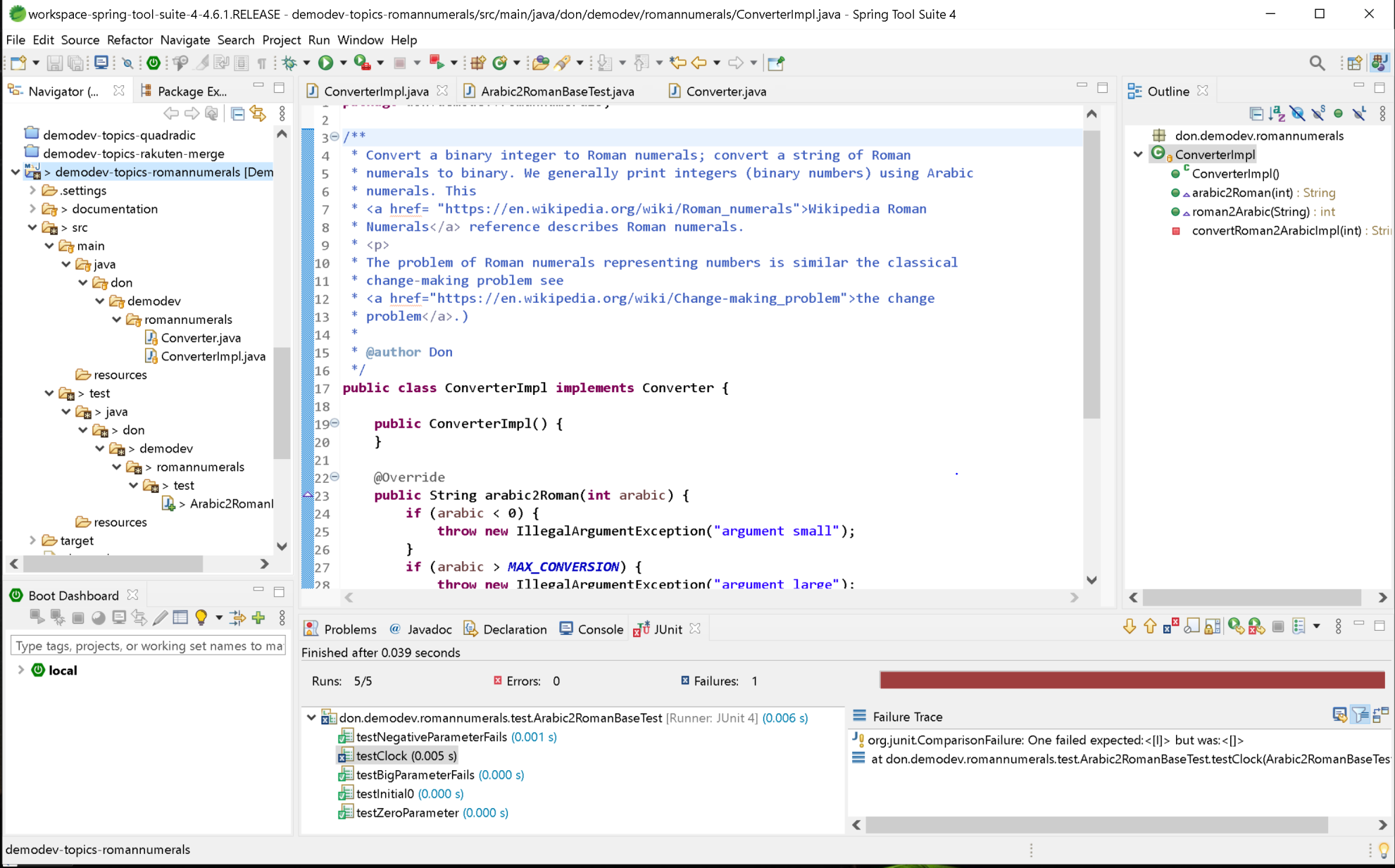
| | \---test

| | 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 use to later modify the implementation in many ways by separating API from implementation. Here is a view of the Eclipse project.



## Initial Tests

We begin test definition by checking preconditions and validations done by the initial implementation. We do this before beginning functionality development with a simple clock time. 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** testClock() {

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

}

}

# Algorithm Development

Reviewing the common greedy algorithm approach for making change, we note that the “largest coins” are used first, with remaining amounts likewise decreased using the next largest coins, until the entire change amount is accumulated. We apply this approach to our converter to implement the *additive* notation. Once we have a basic 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 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 value to be converted using *subtractive* notation. The remaining three rules from reference #4 outline 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 unit tests need to be commented out because the simple additive notation uses more than three consecutive symbols, failing the test. Our initial refactoring, combining additive and subtractive notation, is:

**private** **static** **final** **class** RomanNotation {

**public** **final** String romanNumerals;

**public** **final** **int** subtracted;

**public** RomanNotation(String romanNumerals, **int** subtracted) {

**super**();

**this**.romanNumerals = romanNumerals;

**this**.subtracted = subtracted;

}

}

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

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

**int** numeralIndex = 0;

**int** value = arabic;

**while** (value > 0) {

**final** RomanNotation romanNotation = useAdditiveNotation(value, numeralIndex);

**if** (!romanNotation.romanNumerals.isEmpty()) {

roman.append(romanNotation.romanNumerals);

value -= romanNotation.subtracted;

}

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

**if** (arabic < 1) {

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

}

**return** **null**;

}

We

ccccccccccccccccccccccccccccc

# 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>.