Input Space Partitioning

Assignment Description

For this assignment you will practice designing tests using input space partitioning methods (i.e., interfaced-based and functionality-based).

Your target for this exercise is a system that converts the representation of a number between roman numerals and arabic numerals. The public methods of this system are:

```
String toRoman(Integer arabicNumber);
Integer fromRoman(String romanNumber);
```

For both the interface-based approach and the functionality-based approach, you will do the following:

1. Identify characteristics of the input (Integer and String for interface-based; arabic numbers and roman numbers for functionality-based).
2. Choose a partitioning scheme for each characteristic. The characteristics and corresponding blocks should be documented in your report. Tables, like those shown in the slides, are the preferred way of presenting this information. Your report should also include a justification of why the partitioning schemes result in blocks that are both complete and disjoint.
3. Choose a criterion to generate test requirements. Your choice and rationale should be documented in your report.
4. Implement a test suite that satisfies the criterion using the JUnit testing framework. Note you should keep the tests created by each approach separate (e.g., separate files or separate directories).

Once you have implemented both test suites, you should find least two implementations of the system (e.g., Google: java roman converter). Document where you obtained the systems and any modification you needed to make in your report.

Finally, evaluate the systems by running your test suites, both individually and together. Include a description of the results in your report. Do you now have enough confidence to use either system in a mission critical setting? Explain your opinion in your report.

Identify Characteristics

Interface-Based

String

The StringUtils library offers methods that speak to potential String characteristics:

- isAllLowerCase
  - Checks if the CharSequence contains only lowercase characters.
- isAllUpperCase
  - Checks if the CharSequence contains only uppercase characters.

- isAlpha
    - Checks if the CharSequence contains only Unicode letters.
- isAlphanumeric
    - Checks if the CharSequence contains only Unicode letters or digits.
- isAlphanumericSpace
    - Checks if the CharSequence contains only Unicode letters, digits or space (' ').
- isAlphaSpace
    - Checks if the CharSequence contains only Unicode letters and space (' ').
- isBlank
    - Checks if a CharSequence is empty (""), null or whitespace only.
- isEmpty
    - Checks if a CharSequence is empty ("") or null.
- isMixedCase
    - Checks if the CharSequence contains mixed casing of both uppercase and lowercase characters.
- isNotBlank
    - Checks if a CharSequence is not empty (""), not null and not whitespace only.
- isNotEmpty
    - Checks if a CharSequence is not empty ("") and not null.
- isNumeric
    - Checks if the CharSequence contains only Unicode digits.
- isNumericSpace
    - Checks if the CharSequence contains only Unicode digits or space (' ').
- isWhitespace
    - Checks if the CharSequence contains only whitespace.

Characteristic Table

Characteristic	**b1**	**b2**	**b3**	**b4**
Case	All Lower (C_L)	All Upper (C_U)	Mixed (C_M)	
Contains Number	True (N_T)	False (N_F)		
Contains Symbol	True (S_T)	False (S_F)		
Contains Whitespace	True (W_T)	False (W_F)		
Length	Null (L_N)	Zero (L_0)	One (L_1)	More Than One (L_M)

Justification

From the characteristics outlined in the table, we can construct a String meeting any specific description from the StringUtils methods. From those definitions, any character without an uppercase/lowercase distinction is considered lowercase. Therefore the first characteristic, Case, is complete for all non-empty strings. If a given

String is all lowercase, it can not be all uppercase or mixed-case. The same argument holds for an all uppercase String. If the String has both lowercase and uppercase characters, it is of mixed-case. Therefore, the first characteristic is disjoint for all non-empty strings.

The following two characteristics, Contains Space and Contains Number, are binary properties for any given string. Therefore these two characteristics are complete and disjoint.

The final characteristic, Length, is trivially complete and disjoint with the accommodation that Null be considered an aspect of length. Its inclusion allows Length to serve as a stem when constructing testing criteria.

Integer

An integer's relationship to zero forms three characteristics:

- isNegative
- isZero
- isPositive

It could also be useful to check the min and max Integer values:

- isIntegerMin
- isIntegerMax

Characteristic Table

Characteristic	**b1**	**b2**	**b3**
Relationship to Zero	Less Than Zero	Zero	More Than Zero
Minimum Integer	True	False	
Maximum Integer	True	False	

Justification

The Integers can be separated into three mutually exclusive categories: negative numbers, positive numbers, and zero. Therefore the first characteristic is complete and disjoint.

The second and third characteristics relate to the values held by Integer.MIN_VALUE and Integer.MAX_VALUE. For any given Integer, it either is the value in question or isn't. Therefore these characteristics are complete and disjoint.

Functionality-Based

String

The relevant criteria is whether a given String is a valid Roman Numeral expression. A valid Roman Numeral can be captured with the following characteristics derived from the standard form. As a consequence of standard form, Roman Numerals can only express values on the range [1,4000).

Characteristic Table

Characteristic	**b1**	**b2**
Ones Place Contains At Most One Of {“I”, “II”, “III”, “IV”, “V”, “VI”, “VII”, “VIII”, “IX”}	True	False
Tens Place Contains At Most One Of {“X”, “XX”, “XXX”, “XL”, “L”, “LX”, “LXX”, “LXXX”, “XC”}	True	False
Hundreds Place Contains At Most One Of {“C”, “CC”, “CCC”, “CD”, D”, “DC”, “DCC”, “DCCC”, “CM”}	True	False
Thousands Place Contains At Most One Of {“M”, “MM”, “MMM”}	True	False
Numeral Proceeds From Greatest Decimal Magnitude to Least	True	False

Justification

Each of the “order-of-magnitude” characteristics allows for their occurrence any number of times, including none. Therefore, for any given numeral, these characteristics are complete. However, whether the characteristic occurs at most once is a binary property of the numeral. Consequently, the characteristics are disjoint.

The characteristic that the numeral is ordered from the highest order of magnitude to the lowest is either true or false. Therefore, this characteristic is both complete and disjoint.

Integer

Roman Numerals in standard form are defined on the range [1, 4000). In turn, this provides the primary characteristic for Integers.

Characteristic Table

Characteristic	**b1**	**b2**
Integer Exists On [1, 4000)	True	False

Justification

Whether an Integer exists on the half-open interval [1, 4000) is a binary property. Therefore this characteristic is both complete and disjoint.

Identify Test Criteria

Interface-Based

String

Using Length as the stem of potential test criteria can be constructed using a Cartesian product of the remaining characteristics. This process could be used to construct a String of arbitrary length.

L_N:
[null]

L_0:
[“”]

L_1:
[“C_L”, “C_U”, “N_T”, “S_T”, “W_T”]

L_M:
[$All\ combinations\ of\ L_1\ that\ form\ a\ length\ of\ M$]

For testing, $M = 3$ should be sufficient. This length allows for mixed-case (where C_L and C_U appear together), and for non-case characteristics (Number, Symbol, Whitespace) to appear at the beginning, middle, and end of test cases. Lengths L_1 and L_2 will be excluded as they are subsumed by L_3.

Specific values should be chosen such that all tests result in an exception.
[null]
[“”]
[All length three combinations of: “a”, “A”, “9”, “-“, “ “]

Integer

Testing the boundaries of positive and negative numbers, in addition to zero, should be sufficient. There’s no reason to expect any of the intermediate values to yield different behavior.

The only valid value should be 1. All other values ([Integer.MIN_VALUE, -1, 0, Integer.MAX_VALUE]) should result in an exception.

Functionality-Based

String

We have established characteristics based on units and order. The unit characteristics can be tested individually with their first block (**b1**) indicating a valid numeral, and the second (**b2**) indicating an invalid numeral. Constructing numerals in order will validate the correct combinations of blocks (**b1**), while out-of-order constructions should result in invalid numerals (**b2**)

It should be straightforward to construct arrays for each decimal unit. With one loop we can verify all the unit arrays concurrently. Those arrays can then be used to construct values from (1000, 4000) that will validate correctly ordered numerals.

Invalid numerals should be constructed by hand. The repeating pattern of Roman numerals does not lend itself to iterative construction for invalid values. It should be sufficient to test a handful of numerals, including values with multiple instances of the same decimal unit and disordered numerals. All invalid numerals should result with an exception.

Integer

Testing the integers (1000, 4000) should be sufficient to test the values [1, 4000) as (1000, 4000) includes values from [1, 1000) multiple times.

There are two regions for invalid integers: < 1 and >= 4000. There is no reason to believe that values within these regions are significantly different. Therefore, we can test 0 and 4000. The invalid integers should result with an exception.

Implementations

Implementation A

fromRoman()

Source: https://www.geeksforgeeks.org/converting-roman-numerals-decimal-lying-1-3999/
Modifications:

- Changed method name
- Removed comments

toRoman()

Source: https://algorithms.tutorialhorizon.com/convert-integer-to-roman/
Modifications:

- Changed method name

- Removed output to stdout
- Return constructed string

Implementation B

fromRoman()

Source: https://www.baeldung.com/java-convert-roman-arabic
Modifications:

- Changed method name

toRoman()

Source: https://www.baeldung.com/java-convert-roman-arabic
Modifications:

- Changed method name

Test Results

Implementation A

Interface Tests

Test Summary

8	6	0	0.139s
tests	failures	ignored	duration

25%
successful

Failed tests | Packages | Classes

InterfaceTests. testFromRomanEmpty()
InterfaceTests. testFromRomanInvalidLengthThree()
InterfaceTests. [1] -2147483648
InterfaceTests. [2] -1
InterfaceTests. [3] 0
InterfaceTests. [4] 2147483647

Implementation A suffered from a failure to throw exceptions.

With invalid string input, `fromRoman()` returned zero for the empty string and negative one for everything else. Similarly, `toRoman()` has no mechanism to handle negative values, returns the empty string when given zero, and will construct a numeral for any positive value.

Functionality Tests

Test Summary

12	9	0	0.091s
tests	failures	ignored	duration

25%
successful

Failed tests Packages Classes

FunctionalityTests. testFromRomanInvalidDoubleUnits()
FunctionalityTests. [1] IIIIIV
FunctionalityTests. [2] VX
FunctionalityTests. [3] LC
FunctionalityTests. [4] DM
FunctionalityTests. [5] VL
FunctionalityTests. [6] XD
FunctionalityTests. [7] IM
FunctionalityTests. testToRomanInvalid()

Implementation A's functionality test failures all relate to format checking.

When converting from a Roman numeral, the integer calculation proceeds without checking whether the numeral is well-formed. This occurred when given multiple instances of the same unit (ie "IIIIIV") and disordered numerals (ie "IM").

When converting to a Roman numeral, exceptions were not thrown for values less than one, or above four thousand.

Combined Tests

Test Summary

20	15	0	0.110s
tests	failures	ignored	duration

25%
successful

Failed tests | Packages | Classes

FunctionalityTests. testFromRomanInvalidDoubleUnits()
FunctionalityTests. [1] IIIIIV
FunctionalityTests. [2] VX
FunctionalityTests. [3] LC
FunctionalityTests. [4] DM
FunctionalityTests. [5] VL
FunctionalityTests. [6] XD
FunctionalityTests. [7] IM
FunctionalityTests. testToRomanInvalid()
InterfaceTests. testFromRomanEmpty()
InterfaceTests. testFromRomanInvalidLengthThree()
InterfaceTests. [1] -2147483648
InterfaceTests. [2] -1
InterfaceTests. [3] 0
InterfaceTests. [4] 2147483647

Implementation B

Interface Tests

Test Summary

8	1	0	0.058s
tests	failures	ignored	duration

87% successful

Failed tests Packages Classes

InterfaceTests. testFromRomanEmpty()

Instead of throwing an exception for the empty string, Implementation B returned zero.

Functionality Tests

Test Summary

12	1	0	0.132s
tests	failures	ignored	duration

91% successful

Failed tests Packages Classes

FunctionalityTests. testFromRomanInvalidDoubleUnits()

Implementation B was somewhat resilient to format errors but failed to throw exceptions for repeated units (ie “VV”).

Combined Tests

Test Summary

20	2	0	0.120s
tests	failures	ignored	duration

90%
successful

Failed tests Packages Classes

FunctionalityTests. testFromRomanInvalidDoubleUnits()
InterfaceTests. testFromRomanEmpty()

Conclusion

Additional context would be needed before trusting either implementation in a mission-critical setting.

The implementations were tested against standard form as Wikipedia references it as the format used when,

> *a Roman numeral is considered a legally binding expression of a number, as in U.S. Copyright law (where an "incorrect" or ambiguous numeral may invalidate a copyright claim, or affect the termination date of the copyright period).*

Implementation B could be made trustworthy with minor modifications, but Implementation A likely shouldn't be trusted under any circumstances.