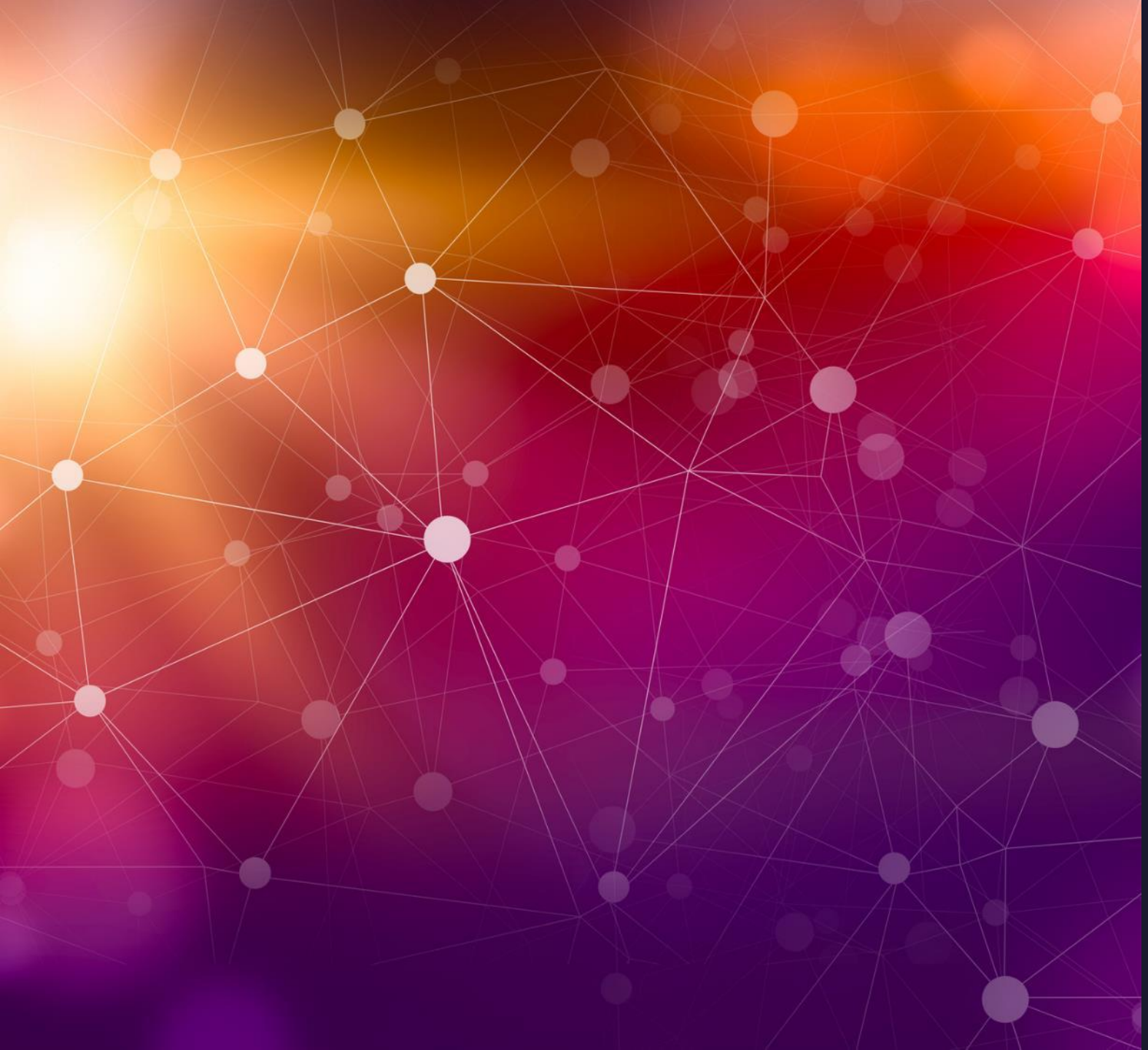

# Transformation Priority Premise

Defining the obvious implementation

# Transformation Priority Premise

The three steps to evolving code in TDD are:

1. Fake it
2. Obvious implementation
3. Triangulation

Faking is self explanatory; just return a value that works

Triangulation is also easy to understand, just write a new failing test, then write code to make it pass

But the obvious implementation is ambiguous since one person in a team may have a different view on what is obvious to another…

…TPP gives us a framework to use in creating the obvious implementation

# What is a **Transformation**?

- Refactoring is changing the structure of code – for readability, performance, maintainability – without *significantly* changing behaviour
- Transformations alter the behaviour without *significantly* changing the structure of the code

**“As the tests get more specific, the code gets more generic.”**

Uncle Bob

Our goal:

to make the code generic and so solve the problem we are presented with

# What about Priority?

({} → null) no code at all → code that returns null

(null → constant)

(constant → constant+) a simple constant to a more complex constant

(constant → scalar) replacing a constant with a variable or an argument

(statement → statements) adding more unconditional statements.

(unconditional → if) splitting the execution path

(scalar → array)

(array → container)

(statement → tail-recursion)

(if → while)

(statement → non-tail-recursion)

(expression → function) replacing an expression with a function or algorithm

(variable → assignment) replacing the value of a variable

# Remember it's a Premise, not a Rule!

It's a guide, not a hard and fast rule

Not all steps have to be employed

Sometimes we might visit steps that are lower in priority first

# Null transformation

Write a test to call a method that does not exist yet. Create the method and have it return null

```
[Test]
● | 0 references
public void return_roman_for_arabic()
{
    var converter = new RomanNumeralConverter();

    var expectedRoman :string = converter.Convert( arabic: 1);

    Assert.That(expectedRoman, Is.EqualTo("I"));
}
```

```
1 reference | ● 0/1 passing
public string Convert(int arabic)
{
    return null;
}
```

# Null -> Constant

Replace the null return value with a constant

```
1 reference | ● 0/1 passing
public string Convert(int arabic)
{
    return "I";
}
```

# Constant -> Constant+

Replace the constant with a more complex constant:

```
[TestCase(arabic: 1, roman: "I")]
[TestCase(arabic: 2, roman: "II")]
● | 0 references
public void return_roman_for_arabic(int arabic, string roman)
{
    var converter = new RomanNumeralConverter();

    var expectedRoman :string = converter.Convert(arabic);

    Assert.That(expectedRoman, Is.EqualTo(roman));
}
```

```
1 reference | ● 0/2 passing
public string Convert(int arabic)
{
    return arabic == 1 ? "I" : "II";
}
```

# Constant -> Scalar

Replace constants with variables:

```
1 reference | ● 0/2 passing
public string Convert(int arabic)
{
    var roman :string = arabic == 1 ? "I" : "II";
    return roman;
}
```

# Statement -> Statements

Add unconditional statements

```
[TestCase(1, "I")]
[TestCase(2, "II")]
[TestCase(2, "III")]
● | 0 references
public void return_roman_for_arabic(int arabic, string roman)
{
    var converter = new RomanNumeralConverter();

    var expectedRoman = converter.Convert(arabic);

    Assert.That(expectedRoman, Is.EqualTo(roman));
}
```

```
1 reference | ● 0/3 passing
public string Convert(int arabic)
{
    var roman = new string(c: 'I', arabic);
    return roman;
}
```

# Unconditional => if

Split the execution path:

```
[TestCase(arabic: 1, roman: "I")]
[TestCase(arabic: 2, roman: "II")]
[TestCase(arabic: 2, roman: "III")]
[TestCase(arabic: 2, roman: "IV")]
● | 0 references
public void return_roman_for_arabic(int arabic, string roman)
{
    var converter = new RomanNumeralConverter();

    var expectedRoman :string = converter.Convert(arabic);

    Assert.That(expectedRoman, Is.EqualTo(roman));
}
```

```
1 reference | ● 0/4 passing
public string Convert(int arabic)
{
    if (arabic == 4)
        return "IV";

    var roman = new string(c: 'I', arabic);
    return roman;
}
```

# Statement -> Array

Use an array to store the data

```
[TestCase(arabic: 1, roman: "I")]
[TestCase(arabic: 2, roman: "II")]
[TestCase(arabic: 2, roman: "III")]
[TestCase(arabic: 2, roman: "IV")]
[TestCase(arabic: 2, roman: "V")]
● | 0 references
public void return_roman_for_arabic(int arabic, string roman)
{
    var converter = new RomanNumeralConverter();

    var expectedRoman :string = converter.Convert(arabic);

    Assert.That(expectedRoman, Is.EqualTo(roman));
}
```

```
1 reference | ● 0/5 passing
public string Convert(int arabic)
{
    var romans = new string[] { "I", "II", "III", "IV", "V" };
    return romans[arabic - 1];
}
```

# Array -> Container

Simplify the implementation by using a container such as a dictionary to match input key to output value

```
1 reference | ● 0/5 passing
public string Convert(int arabic)
{
    var romans = new Dictionary<int, string>
    {
        { 1, "I" },
        { 2, "II" },
        { 3, "III" },
        { 4, "IV" },
        { 5, "V" }
    };
    return romans[arabic];
}
```

# If -> While

we need to jump ahead …. so let’s say we are now here:

```
[TestCase(arabic: 1, roman: "I")]
[TestCase(arabic: 2, roman: "II")]
[TestCase(arabic: 3, roman: "III")]
[TestCase(arabic: 4, roman: "IV")]
[TestCase(arabic: 5, roman: "V")]
[TestCase(arabic: 6, roman: "VI")]
[TestCase(arabic: 7, roman: "VII")]
[TestCase(arabic: 8, roman: "VIII")]
[TestCase(arabic: 9, roman: "IX")]
[TestCase(arabic: 10, roman: "X")]
● | 0 references
public void return_roman_for_arabic(int arabic, string roman)
{
    var converter = new RomanNumeralConverter();

    var expectedRoman :string = converter.Convert(arabic);

    Assert.That(expectedRoman, Is.EqualTo(roman));
}
```

```
1 reference | ● 0/10 passing
public string Convert(int arabic)
{
    var arabicToRoman = new Dictionary<int, string>
    {
        { 1, "I" },
        { 4, "IV" },
        { 5, "V" },
        { 9, "IX" },
        { 10, "X" }
    };

    if (arabic >= 10)
        return $"{arabicToRoman[10]}{new string(arabicToRoman[1][0], count: arabic - 10)}";

    if (arabic >= 9)
        return $"{arabicToRoman[9]}{new string(arabicToRoman[1][0], count: arabic - 9)}";

    if (arabic > 5)
        return $"{arabicToRoman[5]}{new string(arabicToRoman[1][0], count: arabic - 5)}";

    if (arabic < 4)
        return new string(arabicToRoman[1][0], arabic);

    return arabicToRoman[arabic];
}
```

# If -> While

we need to jump ahead …. so let’s say we are now here:

```
[TestCase(arabic: 1, roman: "I")]
[TestCase(arabic: 2, roman: "II")]
[TestCase(arabic: 3, roman: "III")]
[TestCase(arabic: 4, roman: "IV")]
[TestCase(arabic: 5, roman: "V")]
[TestCase(arabic: 6, roman: "VI")]
[TestCase(arabic: 7, roman: "VII")]
[TestCase(arabic: 8, roman: "VIII")]
[TestCase(arabic: 9, roman: "IX")]
[TestCase(arabic: 10, roman: "X")]
● | 0 references
public void return_roman_for_arabic(int arabic, string roman)
{
    var converter = new RomanNumeralConverter();

    var expectedRoman :string = converter.Convert(arabic);

    Assert.That(expectedRoman, Is.EqualTo(roman));
}
```

We can see a lot of duplication; each condition is based on a key from the dictionary

```
1 reference | ● 0/10 passing
public string Convert(int arabic)
{
    var arabicToRoman = new Dictionary<int, string>
    {
        { 1, "I" },
        { 4, "IV" },
        { 5, "V" },
        { 9, "IX" },
        { 10, "X" }
    };

    if (arabic >= 10)
        return $"{arabicToRoman[10]}{new string(arabicToRoman[1][0], count: arabic - 10)}";

    if (arabic >= 9)
        return $"{arabicToRoman[9]}{new string(arabicToRoman[1][0], count: arabic - 9)}";

    if (arabic > 5)
        return $"{arabicToRoman[5]}{new string(arabicToRoman[1][0], count: arabic - 5)}";

    if (arabic < 4)
        return new string(arabicToRoman[1][0], arabic);

    return arabicToRoman[arabic];
}
```

# If -> While

We can evolve from the repeated if statements to a loop, based on the dictionary key:

```
[TestCase(arabic: 1, roman: "I")]
[TestCase(arabic: 2, roman: "II")]
[TestCase(arabic: 3, roman: "III")]
[TestCase(arabic: 4, roman: "IV")]
[TestCase(arabic: 5, roman: "V")]
[TestCase(arabic: 6, roman: "VI")]
[TestCase(arabic: 7, roman: "VII")]
[TestCase(arabic: 8, roman: "VIII")]
[TestCase(arabic: 9, roman: "IX")]
[TestCase(arabic: 10, roman: "X")]
● | 0 references
public void return_roman_for_arabic(int arabic, string roman)
{
    var converter = new RomanNumeralConverter();

    var expectedRoman :string = converter.Convert(arabic);

    Assert.That(expectedRoman, Is.EqualTo(roman));
}
```

```
1 reference | ● 0/10 passing
public string Convert(int arabic)
{
    var arabicToRoman = new Dictionary<int, string>
    {
        { 10, "X" },
        { 9, "IX" },
        { 5, "V" },
        { 4, "IV" },
        { 1, "I" },
    };
    foreach (var key :int in arabicToRoman.Keys)
    {
        if (arabic >= key)
            return $"{arabicToRoman[key]}{new string(arabicToRoman[1][0], count: arabic - key)}";
    }
    return arabicToRoman[arabic];
}
```

# Expression -> Function

Jumping ahead again:

```
[TestCase(arabic: 1, roman: "I")]
[TestCase(arabic: 2, roman: "II")]
[TestCase(arabic: 3, roman: "III")]
[TestCase(arabic: 4, roman: "IV")]
[TestCase(arabic: 5, roman: "V")]
[TestCase(arabic: 6, roman: "VI")]
[TestCase(arabic: 7, roman: "VII")]
[TestCase(arabic: 8, roman: "VIII")]
[TestCase(arabic: 9, roman: "IX")]
[TestCase(arabic: 10, roman: "X")]
[TestCase(arabic: 11, roman: "XI")]
[TestCase(arabic: 12, roman: "XII")]
[TestCase(arabic: 13, roman: "XIII")]
[TestCase(arabic: 14, roman: "XIV")]
[TestCase(arabic: 15, roman: "XV")]
[TestCase(arabic: 16, roman: "XVI")]
[TestCase(arabic: 17, roman: "XVII")]
[TestCase(arabic: 18, roman: "XVIII")]
[TestCase(arabic: 19, roman: "XIX")]
[TestCase(arabic: 20, roman: "XX")]
[TestCase(arabic: 21, roman: "XXI")]
| 0 references
public void return_roman_for_arabic(int arabic, string roman)
{
    var converter = new RomanNumeralConverter();

    var expectedRoman :string = converter.Convert(arabic);

    Assert.That(expectedRoman, Is.EqualTo(roman));
}
```

```
1 reference | 0/21 passing
public string Convert(int arabic)
{
    var arabicToRoman = new Dictionary<int, string>
    {
        { 10, "X" },
        { 9, "IX" },
        { 5, "V" },
        { 4, "IV" },
        { 1, "I" },
    };

    var toReturn = new StringBuilder();
    var arabicNumeral :int = arabic;

    foreach (var (key :int, value :string) in arabicToRoman)
    {
        while (arabicNumeral >= key)
        {
            toReturn.Append(value);
            arabicNumeral -= key;
        }
    }
    return toReturn.ToString();
}
```

# Expression -> Function

Jumping ahead again:

```
[TestCase(arabic: 1, roman: "I")]
[TestCase(arabic: 2, roman: "II")]
[TestCase(arabic: 3, roman: "III")]
[TestCase(arabic: 4, roman: "IV")]
[TestCase(arabic: 5, roman: "V")]
[TestCase(arabic: 6, roman: "VI")]
[TestCase(arabic: 7, roman: "VII")]
[TestCase(arabic: 8, roman: "VIII")]
[TestCase(arabic: 9, roman: "IX")]
[TestCase(arabic: 10, roman: "X")]
[TestCase(arabic: 11, roman: "XI")]
[TestCase(arabic: 12, roman: "XII")]
[TestCase(arabic: 13, roman: "XIII")]
[TestCase(arabic: 14, roman: "XIV")]
[TestCase(arabic: 15, roman: "XV")]
[TestCase(arabic: 16, roman: "XVI")]
[TestCase(arabic: 17, roman: "XVII")]
[TestCase(arabic: 18, roman: "XVIII")]
[TestCase(arabic: 19, roman: "XIX")]
[TestCase(arabic: 20, roman: "XX")]
[TestCase(arabic: 21, roman: "XXI")]
| 0 references
public void return_roman_for_arabic(int arabic, string roman)
{
    var converter = new RomanNumeralConverter();

    var expectedRoman :string = converter.Convert(arabic);

    Assert.That(expectedRoman, Is.EqualTo(roman));
}
```

```
1 reference | 0/21 passing
public string Convert(int arabic)
{
    var arabicToRoman = new Dictionary<int, string>
    {
        { 10, "X" },
        { 9, "IX" },
        { 5, "V" },
        { 4, "IV" },
        { 1, "I" },
    };

    var toReturn = new StringBuilder();
    var arabicNumeral :int = arabic;

    foreach (var (key :int, value :string) in arabicToRoman)
    {
        while (arabicNumeral >= key)
        {
            toReturn.Append(value);
            arabicNumeral -= key;
        }
    }
    return toReturn.ToString();
}
```

Let's extract a method from this

# Expression -> Function

We now have:

```
[TestCase(arabic: 1, roman: "I")]
[TestCase(arabic: 2, roman: "II")]
[TestCase(arabic: 3, roman: "III")]
[TestCase(arabic: 4, roman: "IV")]
[TestCase(arabic: 5, roman: "V")]
[TestCase(arabic: 6, roman: "VI")]
[TestCase(arabic: 7, roman: "VII")]
[TestCase(arabic: 8, roman: "VIII")]
[TestCase(arabic: 9, roman: "IX")]
[TestCase(arabic: 10, roman: "X")]
[TestCase(arabic: 11, roman: "XI")]
[TestCase(arabic: 12, roman: "XII")]
[TestCase(arabic: 13, roman: "XIII")]
[TestCase(arabic: 14, roman: "XIV")]
[TestCase(arabic: 15, roman: "XV")]
[TestCase(arabic: 16, roman: "XVI")]
[TestCase(arabic: 17, roman: "XVII")]
[TestCase(arabic: 18, roman: "XVIII")]
[TestCase(arabic: 19, roman: "XIX")]
[TestCase(arabic: 20, roman: "XX")]
[TestCase(arabic: 21, roman: "XXI")]
0 references
public void return_roman_for_arabic(int arabic, string roman)
{
    var converter = new RomanNumeralConverter();

    var expectedRoman :string = converter.Convert(arabic);

    Assert.That(expectedRoman, Is.EqualTo(roman));
}
```

```
1 reference
public class RomanNumeralConverter
{
    private readonly Dictionary<int, string> arabicToRoman = new()
    {
        { 10, "X" },
        { 9, "IX" },
        { 5, "V" },
        { 4, "IV" },
        { 1, "I" },
    };

    1 reference | 0/21 passing
    public string Convert(int arabic)
    {

        var toReturn = new StringBuilder();

        AppendRomanNumerals(arabicToRoman, arabic, toReturn);

        return toReturn.ToString();
    }

    1 reference
    private static void AppendRomanNumerals(Dictionary<int, string> arabicToRoman,
        int arabicNumeral,
        StringBuilder toReturn)
    {
        foreach (var (key :int , value :string ) in arabicToRoman)
        {
            while (arabicNumeral >= key)
            {
                toReturn.Append(value);
                arabicNumeral -= key;
            }
        }
    }
}
```

# Variable -> Assignment

mutate a variable, i.e. change its value

```
[TestCase(arabic: 1, roman: "I")]
[TestCase(arabic: 2, roman: "II")]
[TestCase(arabic: 3, roman: "III")]
[TestCase(arabic: 4, roman: "IV")]
[TestCase(arabic: 5, roman: "V")]
[TestCase(arabic: 6, roman: "VI")]
[TestCase(arabic: 7, roman: "VII")]
[TestCase(arabic: 8, roman: "VIII")]
[TestCase(arabic: 9, roman: "IX")]
[TestCase(arabic: 10, roman: "X")]
[TestCase(arabic: 11, roman: "XI")]
[TestCase(arabic: 12, roman: "XII")]
[TestCase(arabic: 13, roman: "XIII")]
[TestCase(arabic: 14, roman: "XIV")]
[TestCase(arabic: 15, roman: "XV")]
[TestCase(arabic: 16, roman: "XVI")]
[TestCase(arabic: 17, roman: "XVII")]
[TestCase(arabic: 18, roman: "XVIII")]
[TestCase(arabic: 19, roman: "XIX")]
[TestCase(arabic: 20, roman: "XX")]
[TestCase(arabic: 21, roman: "XXI")]
| 0 references
public void return_roman_for_arabic(int arabic, string roman)
{
    var converter = new RomanNumeralConverter();

    var expectedRoman :string = converter.Convert(arabic);

    Assert.That(expectedRoman, Is.EqualTo(roman));
}
```

```
public class RomanNumeralConverter
{
    private readonly Dictionary<int, string> arabicToRoman = new()
    {
        { 10, "X" },
        { 9, "IX" },
        { 5, "V" },
        { 4, "IV" },
        { 1, "I" },
    };

    1 reference | 0/21 passing
    public string Convert(int arabic)
    {

        var toReturn = new StringBuilder();

        AppendRomanNumerals(arabicToRoman, arabic, toReturn);

        return toReturn.ToString();
    }

    1 reference
    private static void AppendRomanNumerals(Dictionary<int, string> arabicToRoman,
        int arabicNumeral,
        StringBuilder toReturn)
    {
        foreach (var (key :int , value :string ) in arabicToRoman)
        {
            while (
                toReturn.Append(value);
                arabicNumeral -= key;
            }
        }
    }
}
```

# Summary

The key take aways from this for me are that we start simple, and build up our tests (making them more specific) and work our way down the priority to make our code more generic; we don’t have to nor should we always use every step; we will sometimes use transformations out of priority order, but that’s ok, because it’s not a rule, just a premise