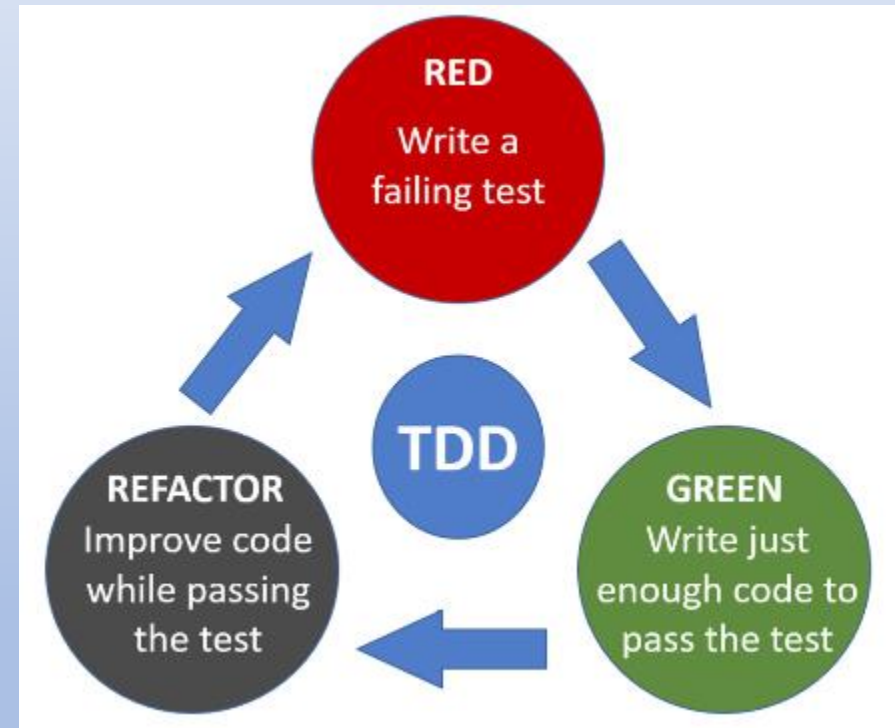


Transformation Priority Premise

a guideline for approaching Test-Driven Development

What is TDD?

- Test-driven development (TDD) ([Beck 2003](#); [Astels 2003](#)), is an eXtreme Programming (XP) practice which combines test-first development, where you write a test before you write just enough production code to fulfill that test, and [refactoring](#).



The Three Laws of TDD

1. You are not allowed to write any production code unless it is to make a failing unit test pass.
2. You are not allowed to write any more of a unit test than is sufficient to fail; and compilation failures are failures.
3. **You are not allowed to write any more production code than is sufficient to pass the one failing unit test.**

Focusing on the third law, the Transformation Priority Premise:

- Guides us on how to apply small evolutions to the code under test
- Helps us avoid *taking big steps/writing complex implementation*
- Is a list of code transformations *ordered by complexity*

Transformation Priority Premise

#	TRANSFORMATION	STARTING CODE	FINAL CODE
1	<code>{}</code> => <code>nil</code>		<code>return nil</code>
2	<code>nil</code> => <code>constant</code>	<code>return nil</code>	<code>return "1"</code>
3	<code>constant</code> => <code>constant+</code>	<code>return "1"</code>	<code>return "1" + "2"</code>
4	<code>constant</code> => <code>scalar</code>	<code>return "1" + "2"</code>	<code>return argument</code>
5	<code>statement</code> => <code>statements</code>	<code>return argument</code>	<code>return arguments</code>
6	<code>unconditional</code> => <code>conditional</code>	<code>return arguments</code>	<code>if(condition) return arguments</code>
7	<code>scalar</code> => <code>array</code>	<code>dog</code>	<code>[dog, cat]</code>
8	<code>array</code> => <code>container</code>	<code>[dog, cat]</code>	<code>{dog = "DOG", cat = "CAT"}</code>
9	<code>statement</code> => <code>tail recursion</code>	<code>a + b</code>	<code>a + recursion</code>
10	<code>conditional</code> => <code>loop</code>	<code>if(condition)</code>	<code>while(condition)</code>
11	<code>tail recursion</code> => <code>full recursion</code>	<code>a + recursion</code>	<code>recursion</code>
12	<code>expression</code> => <code>function</code>	<code>today - birthday</code>	<code>CalculateAge()</code>
13	<code>variable</code> => <code>mutation</code>	<code>day</code>	<code>var day = 10; day = 11;</code>
14	<code>switch case</code>		

Prime Factors kata

1. Made popular by Uncle Bob Martin
2. This kata demonstrates the transformation priority premise
3. Goal is to write a *PrimeFactors generator* that, given an integer, returns the list containing the prime factors in numerical sequence.

- 1 should return `[]`
- 2 should return `[2]`
- 3 should return `[3]`
- 4 should return `[2,2]`
- 5 should return `[5]`
- 6 should return `[2,3]`
- 7 should return `[7]`
- 8 should return `[2,2,2]`
- 9 should return `[3,3]`
- 4620 should return `[2,2,3,5,7,11]`

TPP #1 application (nil constant) ✓

```
[TestMethod]
[DataRow(1, new int[] { })]
public void ReturnPrimeFactorsForNumber(int number, int[] expected)
{
    int[] primes = _primeFactors.Generate(number);

    CollectionAssert.AreEqual(expected, primes);
}
```

```
public class PrimeFactors
{
    public int[] Generate(int number)
    {
        return new int[] { };
    }
}
```

TPP #4 application (constant scalar) X

```
[TestMethod]
[DataRow(1, new int[] { })]
[DataRow(2, new int[] { 2 })]
public void ReturnPrimeFactorsForNumber(int number, int[] expected)
{
    int[] primes = _primeFactors.Generate(number);

    CollectionAssert.AreEqual(expected, primes);
}
```

```
public class PrimeFactors
{
    public int[] Generate(int number)
    {
        return new int[] { number };
    }
}
```


TPP #5 application (statement statements) X

```
[TestMethod]
[DataRow(1, new int[] { })]
[DataRow(2, new int[] { 2 })]
public void ReturnPrimeFactorsForNumber(int number, int[] expected)
{
    int[] primes = _primeFactors.Generate(number);

    CollectionAssert.AreEqual(expected, primes);
}
```

```
public class PrimeFactors
{
    public int[] Generate(int number)
    {
        var primes = new List<int>();
        primes.Add(number);

        return primes.ToArray();
    }
}
```

TPP #6 application (unconditional conditional)

```
[TestMethod]
[DataRow(1, new int[] { })]
[DataRow(2, new int[] { 2 })]
public void ReturnPrimeFactorsForNumber(int number, int[] expected)
{
    int[] primes = _primeFactors.Generate(number);

    CollectionAssert.AreEqual(expected, primes);
}
```

```
public class PrimeFactors
{
    public int[] Generate(int number)
    {
        var primes = new List<int>();

        if (number != 1)
            primes.Add(number);

        return primes.ToArray();
    }
}
```

Add a new failing test

```
[TestMethod]
[DataRow(1, new int[] { })]
[DataRow(2, new int[] { 2 })]
[DataRow(3, new int[] { 3 })]
[DataRow(4, new int[] { 2, 2 })]
public void ReturnPrimeFactorsForNumber(int number, int[] expected)
{
    int[] primes = _primeFactors.Generate(number);

    CollectionAssert.AreEqual(expected, primes);
}
```

TPP #6 application (unconditional conditional) ✓

```
public int[] Generate(int number)
{
    var primes = new List<int>();

    if (number % 2 == 0)
    {
        primes.Add(2);
        number = number / 2;
    }

    if (number != 1)
        primes.Add(number);

    return primes.ToArray();
}
```

Add another failing test

```
[TestMethod]
[DataRow(1, new int[] { })]
[DataRow(2, new int[] { 2 })]
[DataRow(3, new int[] { 3 })]
[DataRow(4, new int[] { 2, 2 })]
[DataRow(5, new int[] { 5 })]
[DataRow(6, new int[] { 2, 3 })]
[DataRow(7, new int[] { 7 })]
[DataRow(8, new int[] { 2, 2, 2| })]
public void ReturnPrimeFactorsForNumber(int number, int[] expected)
{
    int[] primes = _primeFactors.Generate(number);

    CollectionAssert.AreEqual(expected, primes);
}
```

TPP #10 application (conditional loop)

```
public int[] Generate(int number)
{
    var primes = new List<int>();

    while (number % 2 == 0)
    {
        primes.Add(2);
        number = number / 2;
    }

    if (number != 1)
        primes.Add(number);

    return primes.ToArray();
}
```

Added a couple of more tests to force generalization of code

```
[TestMethod]
[DataRow(1, new int[] { })]
[DataRow(2, new int[] { 2 })]
[DataRow(3, new int[] { 3 })]
[DataRow(4, new int[] { 2, 2 })]
[DataRow(5, new int[] { 5 })]
[DataRow(6, new int[] { 2, 3 })]
[DataRow(7, new int[] { 7 })]
[DataRow(8, new int[] { 2, 2, 2 })]
[DataRow(9, new int[] { 3, 3 })]
[DataRow(27, new int[] { 3, 3, 3 })]
[DataRow(25, new int[] { 5, 5 })]
[DataRow(125, new int[] { 5, 5, 5 })]
public void ReturnPrimeFactorsForNumber(int number, int[] expected)
{
    int[] primes = _primeFactors.Generate(number);

    CollectionAssert.AreEqual(expected, primes);
}
```

Rule of Three ➡ Extract code duplication

```
public int[] Generate(int number)
{
    var primes = new List<int>();

    while (number % 2 == 0)
    {
        primes.Add(2);
        number = number / 2;
    }

    while (number % 3 == 0)
    {
        primes.Add(3);
        number = number / 3;
    }

    while (number % 5 == 0)
    {
        primes.Add(5);
        number = number / 5;
    }

    if (number != 1)
        primes.Add(number);

    return primes.ToArray();
}
```


TPP #7 application (scalar array)

```
public class PrimeFactors
{
    static int[] Factors = new[] { 2, 3, 5 };

    public int[] Generate(int number)
    {
        var primes = new List<int>();

        foreach (int factor in Factors)
        {
            while (number % factor == 0)
            {
                primes.Add(factor);
                number = number / factor;
            }
        }

        if (number != 1)
            primes.Add(number);

        return primes.ToArray();
    }
}
```

Add another failing test

```
[TestMethod]
[DataRow(1, new int[] { })]
[DataRow(2, new int[] { 2 })]
[DataRow(3, new int[] { 3 })]
[DataRow(4, new int[] { 2, 2 })]
[DataRow(5, new int[] { 5 })]
[DataRow(6, new int[] { 2, 3 })]
[DataRow(7, new int[] { 7 })]
[DataRow(8, new int[] { 2, 2, 2 })]
[DataRow(9, new int[] { 3, 3 })]
[DataRow(27, new int[] { 3, 3, 3 })]
[DataRow(25, new int[] { 5, 5 })]
[DataRow(125, new int[] { 5, 5, 5 })]
[DataRow(4620, new int[] { 2, 2, 3, 5, 7, 11| })]
public void ReturnPrimeFactorsForNumber(int number, int[] expected)
{
    int[] primes = _primeFactors.Generate(number);

    CollectionAssert.AreEqual(expected, primes);
}
```

Update **Factors** array to make test pass ✓

```
public class PrimeFactors
{
    static int[] Factors = new[] { 2, 3, 5, 7, 11 };

    public int[] Generate(int number)
    {
        var primes = new List<int>();

        foreach (int factor in Factors)
        {
            while (number % factor == 0)
            {
                primes.Add(factor);
                number = number / factor;
            }
        }

        if (number != 1)
            primes.Add(number);

        return primes.ToArray();
    }
}
```

That's All Folks!