

Prime (re)Factoring (with AI and ChatGPT!)

Steve Smith

@ardalis | YouTube.com/ardalis

steve@nimblepros.com | NimblePros.com





A bit of math

- Anything multiplied by 1 is itself (identity)
- Multiplying two numbers together produces a **product**
- The two numbers are called **factors**
- Any number can be **factored**, revealing the factors that can be used to produce it via multiplication



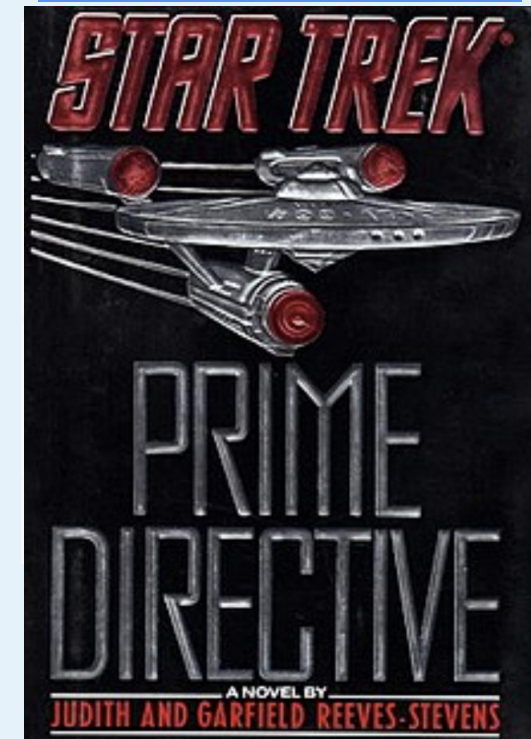
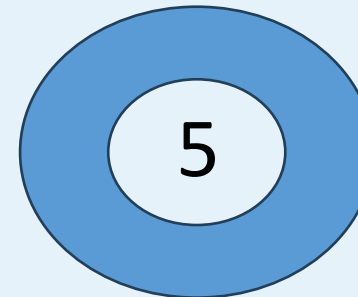
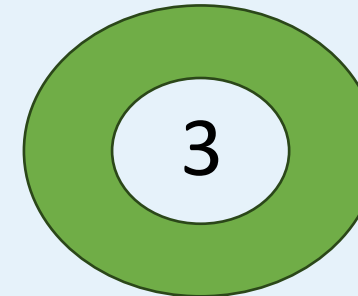
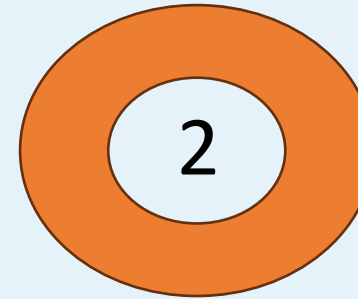
Factors

- The number 2 has factors 1 and 2
- The number 3 has factors 1 and 3
- The number 4 has factors 1 and 4 and 2 and 2
- The number 5 has factors 1 and 5
- The number 6 has factors 1 and 6 and 2 and 3
- Let's toss out the **identity factors** since they're less interesting
 - In fact in math this is frequently done to produce **prime numbers** – numbers which have no factors once the identity factors are ignored



Primes and Colors

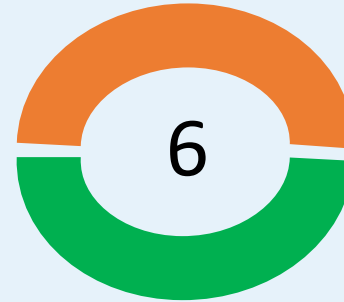
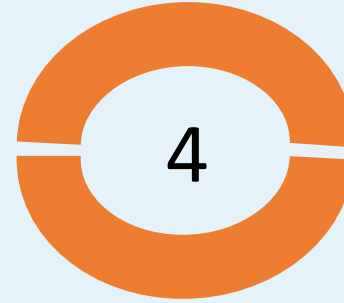
- The number 2 is prime
- The number 3 is prime
- The number 5 is prime





Prime Factors

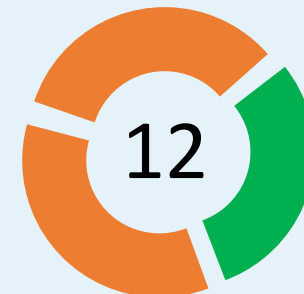
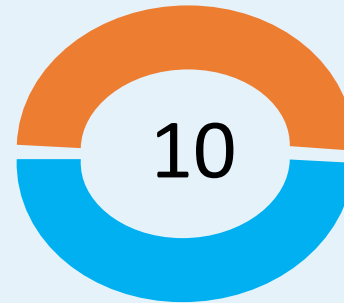
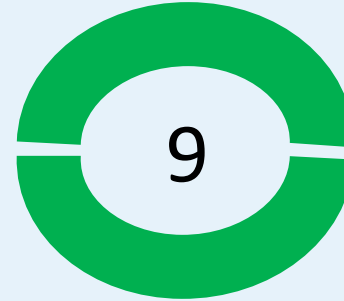
- The number 4 has 2 prime factors
 - 2, 2
- The number 6 has 2 prime factors
 - 2, 3
- The number 8 has 3 prime factors
 - 2, 2, 2



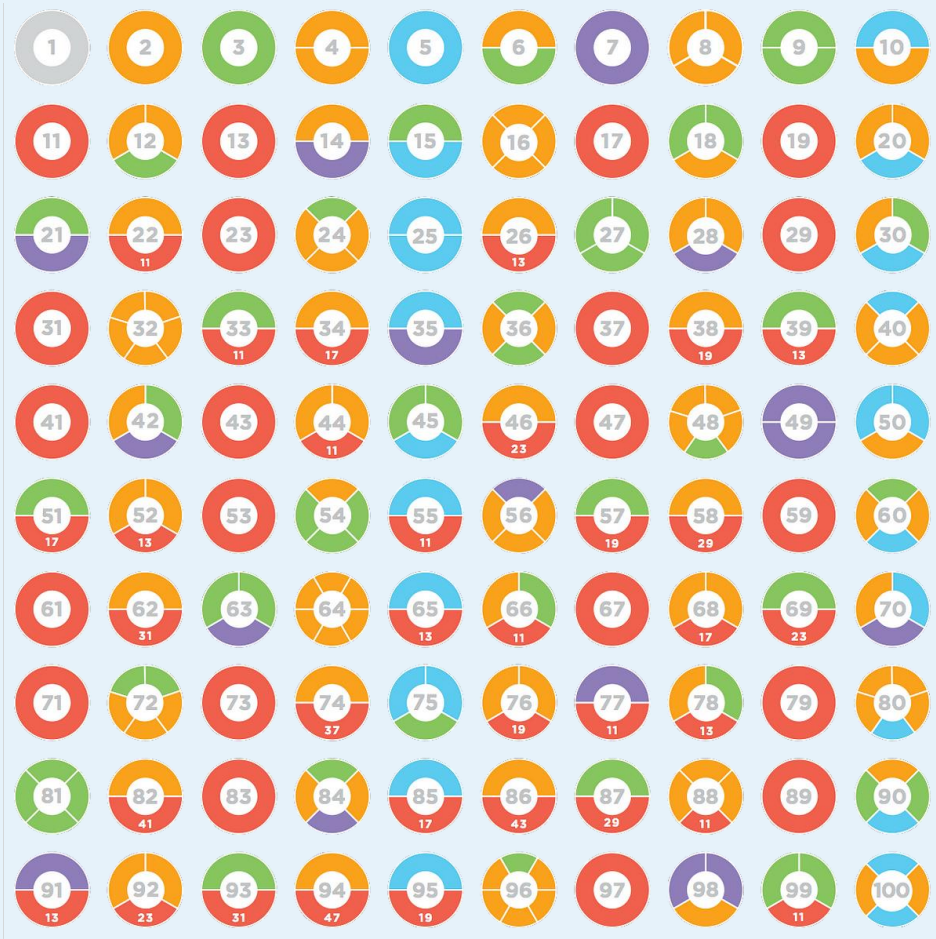


Prime Factors

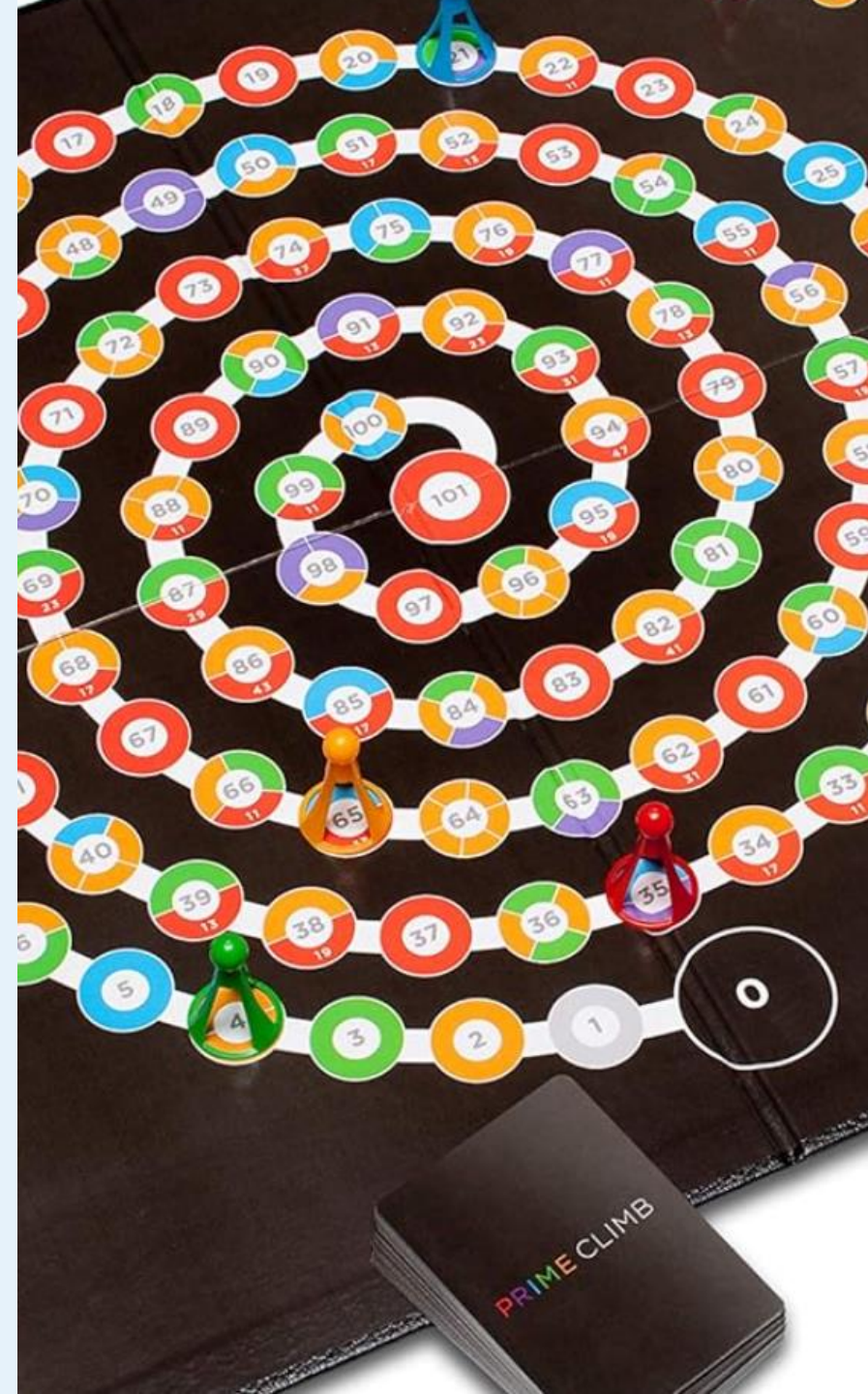
- The number 9 has 2 prime factors
 - 3, 3
- The number 10 has 2 prime factors
 - 2, 5
- The number 12 has 3 prime factors
 - 2, 2, 3



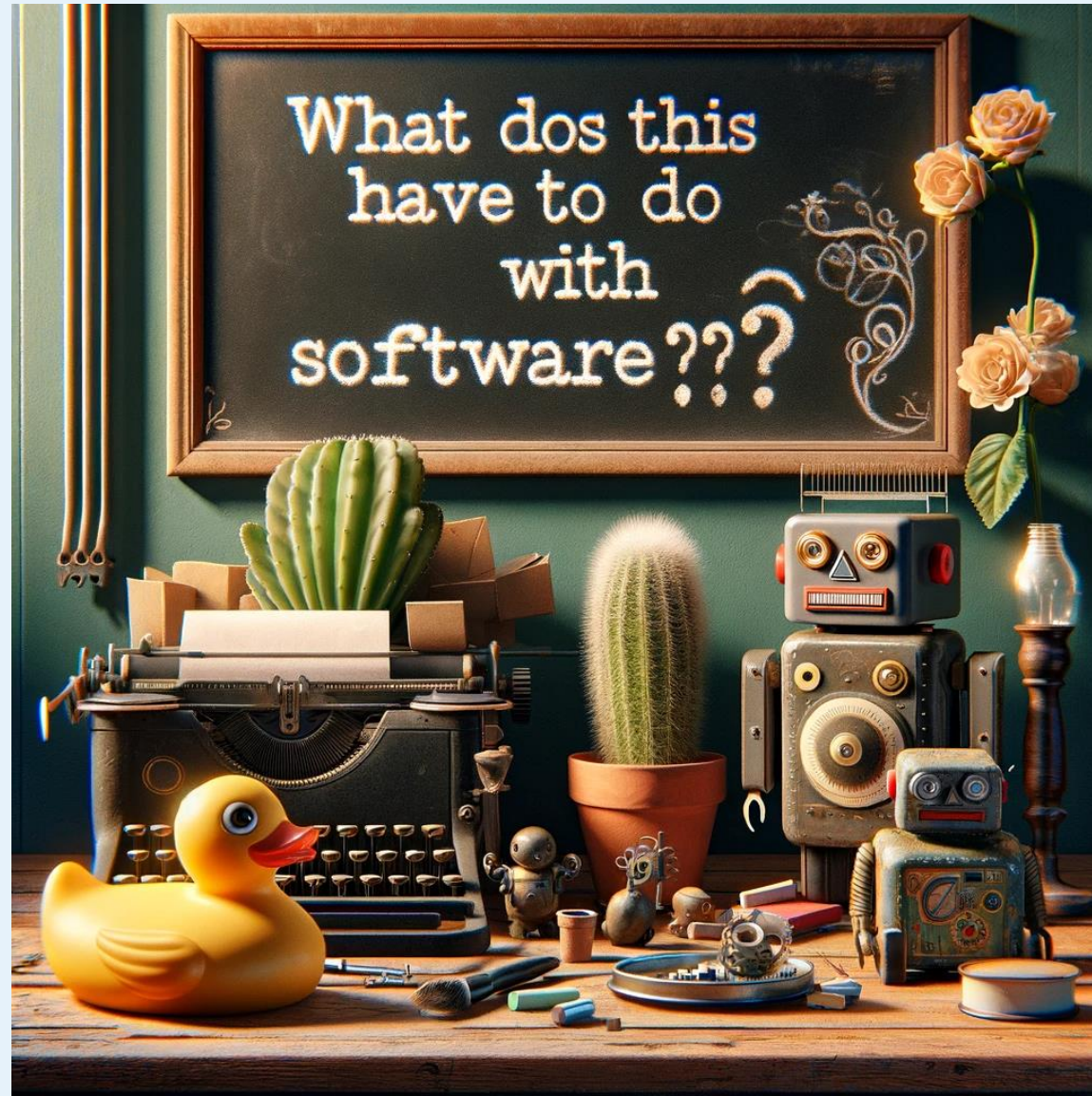
Prime Climb (boardgame)



<https://amzn.to/3RKTeHD>



What does this have to do with software???



Factoring Code

“**Decomposition** in [computer science](#), also known as **factoring**, is breaking a complex [problem](#) or [system](#) into parts that are easier to conceive, understand, program, and maintain.”





SEPARATION OF CONCERNS

Don't let your plumbing code pollute your software.



Common **Factors** or Concerns

- Validation
- Error Handling
- Logging
- Data Access
- Business Logic
- UI Logic
- Authorization
- *and many others*



What if we apply the
mathematical concept of
prime factors to our software?



We can even use Prime Climb's coloring concept to make things easier to visualize

```
public async Task CreateOrder(Cart cart, Customer customer)
{
    try
    {
        Log("Starting order creation.");

        ValidateCart(cart);
        ValidateCustomer(customer);

        Order newOrder = ProcessCart(cart, customer);

        await dbContext.Orders.AddAsync(newOrder);
        await _dbContext.SaveChangesAsync();

        await SendOrderConfirmationEmail(customer.Email);

        UpdateUI("Order created successfully.");
    }
    catch (Exception ex)
    {
        LogError("Error in CreateOrder: " + ex.Message);
        UpdateUI("An error occurred while creating the order.");
        // Additional error handling logic here
    }
}
```


Demo – Identifying and Refactoring

With some help from ChatGPT





ChatGPT Can Classify and Format Code



ChatGPT

Great! I'll format the provided C# code according to the rules you've specified. I'll use different classes for comments (``comment``), mechanical C# code (``mechanical``), UI-related code (``ui``), validation logic (``validation``), logging (``logging``), and business logic (``business``). Here's how the code will look:

html

 Copy code

```
<!DOCTYPE html>

<html>

<head>

  <title>C# Code Block</title>

  <style>

    pre {

      font-family: monospace;

      line-height: 1;

    }

  </style>

</head>

<body>

  <pre>
  </pre>

</body>

</html>
```



ChatGPT Can Classify and Format Code

```
.business {  
    color: black;  
    background-color: #ee82ee; /* Violet for business logic */  
}  
</style>  
</head>  
<body>  
    <pre>  
<span class="mechanical">using Newtonsoft.Json;</span>  
<span class="mechanical">using Newtonsoft.Json.Converters;</span>  
<span class="mechanical">using System;</span>  
<span class="mechanical">using System.IO;</span>  
  
<span class="mechanical">namespace ArdalisRating</span>  
<span class="mechanical">{</span>  
<span class="comment">    /// <summary></span>  
<span class="comment">    /// The RatingEngine reads the policy applica  
<span class="comment">    /// rating value based on the details.</span>  
<span class="comment">    /// </summary></span>  
<span class="mechanical">    public class RatingEngine</span>  
<span class="mechanical">    {</span>  
<span class="mechanical">        public decimal Rating { get; set; }</s  
<span class="mechanical">        public void Rate()</span>  
<span class="mechanical">    {</span>
```



Sometimes it gets distracted though...



ChatGPT



Here's the continuation of your C# code formatting for lines 151-200:

html

 Copy code

```
<pre class="show-error-handling show-logging show-validation show-ui sh
```



Sometimes it gets distracted though...



You

Nice picture you sent but I'm curious why you included that in addition to the formatted code?



ChatGPT

I apologize for the confusion; the inclusion of the image was unintentional and not related to your request for code formatting. Thank you for pointing it out, and if you have more lines of code to format or any other queries, feel free to ask!

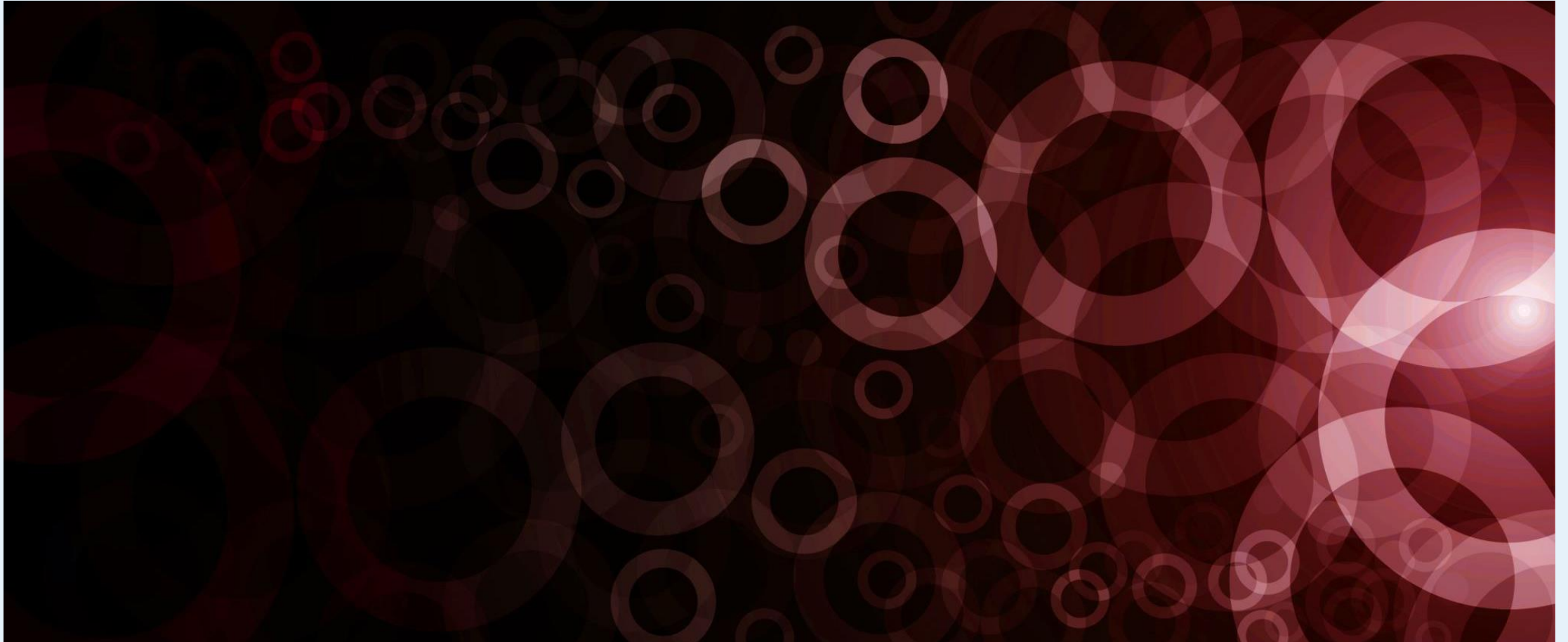
Refactoring for Performance

Finding primes





Find Prime Factors Given Input N





Rules of Simple Design (via Kent Beck)

- Make it work
- Make it right
- Make it fast

<https://wiki.c2.com/?MakeItWorkMakeItRightMakeItFast>



IsPrime() – Version 1



```
public static bool IsPrime1(int number)
{
    if (number < 2) return false;

    for (int i = 2; i < number; i++)
    {
        if (number % i == 0) return false;
    }
    return true;
}
```



It works! We can easily benchmark it, too

```
using BenchmarkDotNet.Attributes;
using PrimeFactorsKata;

[MemoryDiagnoser]
1 reference | 0 changes | 0 authors, 0 changes
public class PrimeBenchmark
{
    public int Limit = 100_000;


    [Benchmark (Description = "Divide by every number less than n")]
    0 references | 0 changes | 0 authors, 0 changes
    public void FindPrimesBenchmark1()
    {
        foreach(int prime in PrimeUtilities.FindPrimesLessThan(Limit, PrimeUtilities.IsPrime1))
        {
            // do nothing
        }
    }
}
```




Benchmark – IsPrime() Version 1

```
// * Summary *
```

```
BenchmarkDotNet v0.13.12, Windows 10 (10.0.19045.3803/22H2/2022Update)
Intel Core i9-9900K CPU 3.60GHz (Coffee Lake), 1 CPU, 16 logical and 8 physical cores
.NET SDK 8.0.100
[Host]      : .NET 8.0.0 (8.0.23.53103), X64 RyuJIT AVX2
DefaultJob  : .NET 8.0.0 (8.0.23.53103), X64 RyuJIT AVX2
```



Method	Mean	Error	StdDev	Allocated
-----	-----	-----	-----	-----
'Divide by every number less than n'	792.8 ms	9.84 ms	8.72 ms	456 B

```
// * Hints *
```

```
Outliers
PrimeBenchmark.'Divide by every number less than n': Default -> 1 outlier was removed (822.36 ms)
```

```
// * Legends *
```

```
Mean      : Arithmetic mean of all measurements
Error      : Half of 99.9% confidence interval
StdDev     : Standard deviation of all measurements
Allocated  : Allocated memory per single operation (managed only, inclusive, 1KB = 1024B)
```



But wait, we don't need to check EVERY number less than the one we're checking

If we check 100

When we divide by 2, we get 50. But if we divide by 50 we get 2.

Factors come in pairs!

So since we're always dividing by 2, we can just go up to $n/2$!



Let's Optimize It

- We don't actually need to check every number less than n
- We can just go up to $n/2$!



IsPrime() – Version 2



```
public static bool IsPrime2(int number)
{
    if (number < 2) return false;

    for (int i = 2; i <= number / 2; i++)
    {
        if (number % i == 0) return false;
    }
    return true;
}
```



New Benchmark Results!

```
// * Summary *
```

```
BenchmarkDotNet v0.13.12, Windows 10 (10.0.19045.3803/22H2/2022Update)
Intel Core i9-9900K CPU 3.60GHz (Coffee Lake), 1 CPU, 16 logical and 8 physical cores
.NET SDK 8.0.100
[Host]      : .NET 8.0.0 (8.0.23.53103), X64 RyuJIT AVX2
DefaultJob  : .NET 8.0.0 (8.0.23.53103), X64 RyuJIT AVX2
```

Method	Mean	Error	StdDev	Ratio	RatioSD	Allocated	Alloc Ratio
'Divide by every number less than n'	797.5 ms	14.77 ms	14.51 ms	1.00	0.00	456 B	1.00
'Divide by every number less than n/2'	398.5 ms	7.22 ms	6.75 ms	0.50	0.02	456 B	1.00

```
// * Legends *
```

```
Mean      : Arithmetic mean of all measurements
Error     : Half of 99.9% confidence interval
StdDev    : Standard deviation of all measurements
Ratio     : Mean of the ratio distribution ([Current]/[Baseline])
RatioSD   : Standard deviation of the ratio distribution ([Current]/[Baseline])
Allocated : Allocated memory per single operation (managed only, inclusive, 1KB = 1024B)
Alloc Ratio : Allocated memory ratio distribution ([Current]/[Baseline])
1 ms      : 1 Millisecond (0.001 sec)
```




Let's Optimize It, Again!

- We don't actually need to check every number up to $n/2$
- We can just go up to \sqrt{n} !
 - Any factors higher than the square root must have a factor less than the square root that we already would have found
- Micro-Optimization!
 - Checking if $i*i$ is less than n is computationally faster than performing \sqrt{n}
 - Let's benchmark to see if that matters, though



IsPrime() – Version 3 (using sqrt)



```
public static bool IsPrime3_Sqrt1(int number)
{
    if (number < 2) return false;

    int limit = (int)Math.Sqrt(number);
    for (int i = 2; i <= limit; i++)
    {
        if (number % i == 0) return false;
    }
    return true;
}
```



IsPrime() – Version 3 (using $i*i$)



```
public static bool IsPrime3_Sqrt2(int number)
{
    if (number < 2) return false;

    for (int i = 2; i * i <= number; i++)
    {
        if (number % i == 0) return false;
    }
    return true;
}
```



New Benchmark Results!

// * Summary *

BenchmarkDotNet v0.13.12, Windows 10 (10.0.19045.3803/22H2/2022Update)
Intel Core i9-9900K CPU 3.60GHz (Coffee Lake), 1 CPU, 16 logical and 8 physical cores
.NET SDK 8.0.100
[Host] : .NET 8.0.0 (8.0.23.53103), X64 RyuJIT AVX2
DefaultJob : .NET 8.0.0 (8.0.23.53103), X64 RyuJIT AVX2

Method	Mean	Error	StdDev	Ratio	Allocated	Alloc Ratio
'Divide by every number less than n'	794.135 ms	11.9365 ms	11.1654 ms	1.000	456 B	1.00
'Divide by every number less than n/2'	392.072 ms	1.5901 ms	1.2414 ms	0.495	456 B	1.00
'Divide by every number less than or equal to sqrt(n)'	5.401 ms	0.0779 ms	0.0729 ms	0.007	59 B	0.13
'Divide by every number less than or equal to sqrt(n) using i*i'	5.325 ms	0.0728 ms	0.0681 ms	0.007	59 B	0.13

// * Hints *

Outliers

PrimeBenchmark.'Divide by every number less than n/2': Default -> 3 outliers were removed (399.09 ms..411.98 ms)

// * Legends *

Mean : Arithmetic mean of all measurements
Error : Half of 99.9% confidence interval
StdDev : Standard deviation of all measurements
Ratio : Mean of the ratio distribution ([Current]/[Baseline])
Allocated : Allocated memory per single operation (managed only, inclusive, 1KB = 1024B)
Alloc Ratio : Allocated memory ratio distribution ([Current]/[Baseline])
1 ms : 1 Millisecond (0.001 sec)



Let's Optimize It, Again! Because we can!

- We don't actually need to use division!
- We can just use addition which is way faster, and keep track of a set of primes for future use
- Known as the Sieve of Eratosthenes
 - Create an array of numbers **2** to **limit**
 - Starting with 2, keep adding that number and marking the results as non-prime
 - Repeat with the next unmarked number until done




Sieve of Eratosthenes

	2	3	4	5	6	7	8	9	10	Prime numbers
11	12	13	14	15	16	17	18	19	20	
21	22	23	24	25	26	27	28	29	30	
31	32	33	34	35	36	37	38	39	40	
41	42	43	44	45	46	47	48	49	50	
51	52	53	54	55	56	57	58	59	60	
61	62	63	64	65	66	67	68	69	70	
71	72	73	74	75	76	77	78	79	80	
81	82	83	84	85	86	87	88	89	90	
91	92	93	94	95	96	97	98	99	100	
101	102	103	104	105	106	107	108	109	110	
111	112	113	114	115	116	117	118	119	120	



Benchmarking with the Sieve



```
public static IEnumerable<int> SieveOfEratosthenes(int limit)
{
    bool[] isPrime = new bool[limit + 1];
    for (int i = 2; i <= limit; i++) isPrime[i] = true;

    for (int p = 2; p * p <= limit; p++)
    {
        if (isPrime[p])
        {
            for (int i = p * p; i <= limit; i += p)
            {
                isPrime[i] = false;
            }
        }
    }

    for (int p = 2; p <= limit; p++)
    {
        if (isPrime[p])
        {
            yield return p;
        }
    }
}
```



New Benchmark Results!

```
// * Summary *
```

```
BenchmarkDotNet v0.13.12, Windows 10 (10.0.19045.3803/22H2/2022Update)
Intel Core i9-9900K CPU 3.60GHz (Coffee Lake), 1 CPU, 16 logical and 8 physical cores
.NET SDK 8.0.100
[Host]      : .NET 8.0.0 (8.0.23.53103), X64 RyuJIT AVX2
DefaultJob  : .NET 8.0.0 (8.0.23.53103), X64 RyuJIT AVX2
```

Method	Mean	Error	StdDev	Ratio	Gen0	Gen1	Gen2	Allocated	Alloc Ratio
'Divide by every number less than n'	794,083.9 us	11,595.18 us	10,278.83 us	1.000	-	-	-	456 B	1.00
'Divide by every number less than n/2'	393,681.7 us	2,517.64 us	1,965.61 us	0.495	-	-	-	456 B	1.00
'Divide by every number less than or equal to sqrt(n)'	5,387.6 us	84.96 us	79.47 us	0.007	-	-	-	59 B	0.13
'Divide by every number less than or equal to sqrt(n) using i*i'	5,347.7 us	78.39 us	73.32 us	0.007	-	-	-	59 B	0.13
'Sieve of Eratosthenes'	410.5 us	8.19 us	8.77 us	0.001	30.7617	30.7617	30.7617	231490 B	507.65

```
// * Hints *
```

```
Outliers
```

```
PrimeBenchmark.'Divide by every number less than n': Default -> 1 outlier was removed (829.28 ms)
PrimeBenchmark.'Divide by every number less than n/2': Default -> 3 outliers were removed (405.90 ms..417.22 ms)
```

```
// * Legends *
```

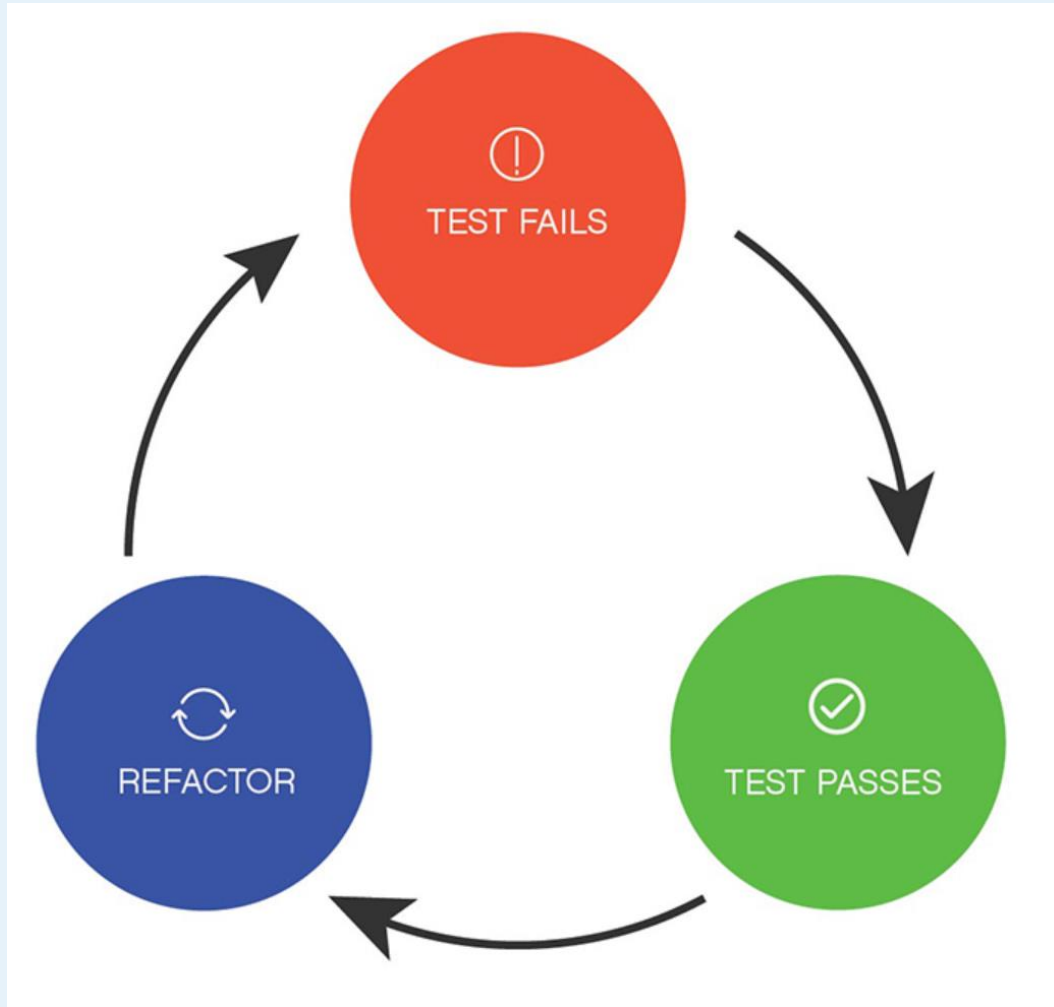
```
Mean      : Arithmetic mean of all measurements
Error     : Half of 99.9% confidence interval
StdDev    : Standard deviation of all measurements
Ratio     : Mean of the ratio distribution ([Current]/[Baseline])
Gen0      : GC Generation 0 collects per 1000 operations
Gen1      : GC Generation 1 collects per 1000 operations
Gen2      : GC Generation 2 collects per 1000 operations
Allocated : Allocated memory per single operation (managed only, inclusive, 1KB = 1024B)
Alloc Ratio : Allocated memory ratio distribution ([Current]/[Baseline])
```

“Great, now we’re ready to find prime factors!”





We'll use TDD to make sure it's right



<http://butunclebob.com/ArticleS.UncleBob.ThePrimeFactorsKata>



The First Test

```
public class PrimeFactors_Generate
{
    [Fact]
    public void ReturnsNoFactorsGiven1()
    {
        var result =
        PrimeFactors.Generate(1);
        Assert.Empty(result);
    }
}
```

```
public static class PrimeFactors
{
    public static List<int> Generate(int
number)
    {
        return new List<int>();
    }
}
```

All Tests Pass



The Second Test (2)

```
[Fact]
public void ReturnsNoFactorsGiven1()
{
    var result =
    PrimeFactors.Generate(1);
    Assert.Empty(result);
}

[Fact]
public void Returns2Given2()
{
    var result =
    PrimeFactors.Generate(2);
    Assert.Contains(result, x => x == 2);
}
```

```
public static class PrimeFactors
{
    public static List<int> Generate(int
number)
    {
        return new List<int>();
    }
}
```

Tests Pass



The Third Test (3)

```
[Fact]
public void ReturnsNoFactorsGiven1()
{
    var result =
        PrimeFactors.Generate(1);
    Assert.Empty(result);
}

[Fact]
public void Returns2Given2()
{
    var result =
        PrimeFactors.Generate(2);
    Assert.Contains(result, x => x == 2);
}

[Fact]
public void Returns3Given3()
{
    var result =
        PrimeFactors.Generate(3);
    Assert.Contains(result, x => x == 3);
}
```

```
public static List<int> Generate(int
number)
{
    List<int> factors = [];
    if(number > 1)
    {
        factors.Add(2); // hardcoded!
    }
    return factors;
}
```

All Tests Pass



As tests get more specific (more test cases), the code under test gets more generic.



The Fourth Test (4)

```
public void ReturnsNoFactorsGiven1()
{
    var result =
    PrimeFactors.Generate(1);
    Assert.Empty(result);
}

[Fact]
public void Returns2Given2()
{
    var result =
    PrimeFactors.Generate(2);
    Assert.Contains(result, x => x == 2);
}

[Fact]
public void Returns3Given3()
{
    var result =
    PrimeFactors.Generate(3);
    Assert.Contains(result, x => x == 3);
}

[Fact]
public void Returns2and2Given4()
{
    var result =
    PrimeFactors.Generate(4);
```

```
public static List<int> Generate(int
number)
{
    List<int> factors = [];
    if ((number % 2 == 0))
    {
        factors.Add(2);
        number /= 2;
    }
    if (number > 1)
    {
        factors.Add(number);
    }
    return factors;
}
```

All Tests Pass



The Fifth Test (6)



```
[Fact]
public void Returns2and2Given4()
{
    var result =
    PrimeFactors.Generate(4);
    Assert.Equal([2,2], result);
}

[Fact]
public void Returns2and3Given6()
{
    var result =
    PrimeFactors.Generate(6);
    Assert.Equal([2,3], result);
}
```



```
public static List<int> Generate(int
number)
{
    List<int> factors = [];
    if ((number % 2 == 0))
    {
        factors.Add(2);
        number /= 2;
    }
    if (number > 1)
    {
        factors.Add(number);
    }
    return factors;
}
```

All Tests Pass



The Sixth Test (8)

```
[Fact]
public void Returns2and3Given6()
{
    var result =
        PrimeFactors.Generate(6);
    Assert.Equal([2,3], result);
}

[Fact]
public void Returns2and2and2Given8()
{
    var result =
        PrimeFactors.Generate(8);
    Assert.Equal([2,2,2], result);
}
```

```
public static List<int> Generate(int
number)
{
    List<int> factors = [];
    while ((number % 2 == 0))
    {
        factors.Add(2);
        number /= 2;
    }
    if (number > 1)
    {
        factors.Add(number);
    }
    return factors;
}
```

All Tests Pass



The Seventh Test (9)

```
[Fact]
public void Returns2and2and2Given8()
{
    var result =
        PrimeFactors.Generate(8);
    Assert.Equal([2,2,2], result);
}

[Fact]
public void Returns3and3Given9()
{
    var result =
        PrimeFactors.Generate(9);
    Assert.Equal([3,3], result);
}
```

```
public static List<int> Generate(int
number)
{
    List<int> factors = [];
    int candidate = 2;
    while (number > 1)
    {
        while (number % candidate == 0)
        {
            factors.Add(candidate);
            number /= candidate;
        }
        candidate++;
    }
    return factors;
}
```

All Tests Pass



We're done!



But, wait...

We didn't actually need `IsPrime` anywhere.



Performance Optimization Tip

The fastest code is the code you don't call at all.



Summary

- Consider different **factors** or concerns in your software
- Leverage AI for categorizing, formatting (small) blocks of code
- Strive to make individual code modules **prime**, refactored down to a single (or at least **primary**) responsibility
- Don't dive into premature optimization!
- Let tests guide your design – you may not actually need that thing you're trying to optimize!



More Resources

- Find me: ardalis.com | steve@nimblepros.com
- Learn more:
 - <https://nimblepros.com/>
 - <https://www.pluralsight.com/authors/steve-smith>