Statement Code Smells and Refactorings

Steve Smith
FORCE MULTIPLIER FOR DEV TEAMS
@ardalis | ardalis.com | weeklydevtips.com

Objectives

Learn various low-level code smells

Learn refactoring techniques to address them

Smell: Primitive Obsession

Overuse of primitives, instead of better abstractions or data structures, results in excess code required to enforce constraints.

Bloater

Example: Primitive Obsession

```
AddHoliday(7,4);
```

Example: Primitive Obsession

```
AddHoliday(7,4);
```

```
Date independenceDay = new Date(7,4);
AddHoliday(independenceDay);
```

Example: Primitive Obsession

```
AddHoliday(7,4);



Date independenceDay = new Date(7,4);
AddHoliday(independenceDay);

Date independenceDay = new Date(month: 7, day: 4);
AddHoliday(independenceDay);
```

Using Constant Values

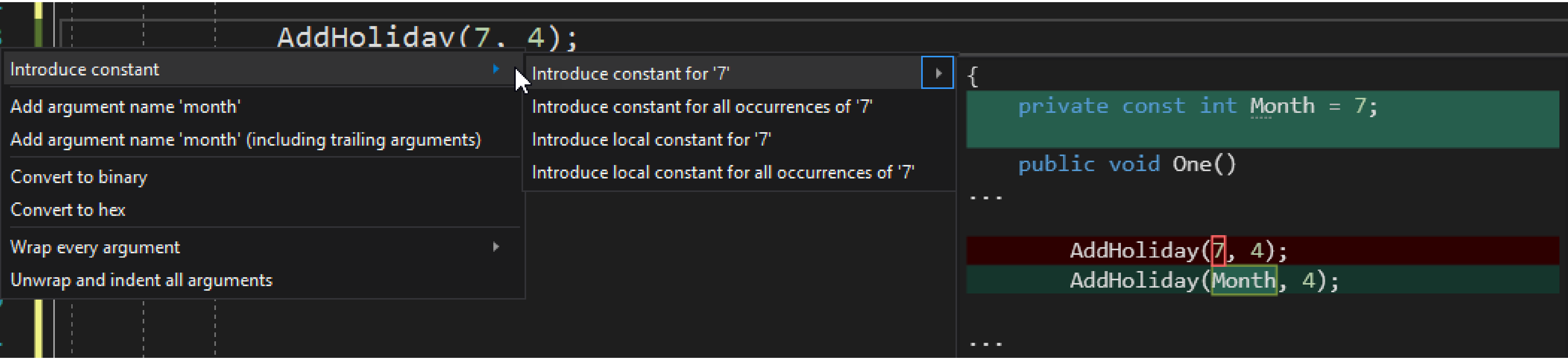
```
AddHoliday(Constants.Month.JULY, Constants.Day.DAY_4);
```

Using Constant Values

```
public static class Constants
{
    public static class Month
    {
        public const int JANUARY = 1;
        public const int FEBRUARY = 2;
        // etc.
    }
}
```

https://ardalis.com/group-your-constants-and-enums

Visual Studio Tooling Assistance



```
AddHoliday(7, 4);
```

Introduce constant
Add argument name 'month'
Add argument name 'month' (including trailing arguments)
Convert to binary
Convert to hex
Wrap every argument
Unwrap and indent all arguments
Introduce constant for '7'
Introduce constant for all occurrences of '7'
Introduce local constant for '7'
Introduce local constant for all occurrences of '7'

```
{
    private const int Month = 7;

    public void One()
...

        AddHoliday(7, 4);
        AddHoliday(Month, 4);

...
```

Using Enums

```
public enum Month
{
    January = 1,
    // etc.
}


public void AddHolidayEnum(Enums.Month month, Enums.Day
day)
{}
```

Client Code with Enums

```
// enums
AddHolidayEnum(Enums.Month.January, Enums.Day.Day_4);


// but also enums
AddHolidayEnum(0, 0);
AddHolidayEnum((Enums.Month)13, (Enums.Day)32);
```

SmartEnum

```
// install Ardalis.SmartEnum package

public sealed class MonthEnum : SmartEnum<MonthEnum>

{

    public static readonly MonthEnum January = new
MonthEnum(nameof(January), 1, "Jan");

    // other months

    public string ShortName { get; set; }

}

// usage

AddHolidaySmartEnum(MonthEnum.July, DayEnum.Day_4);
```

Refactoring Statement Primitive Obsession

Introduce Named Variable

Use Named Arguments

Replace Primitive with Constant

Replace Primitive with Enum/SmartEnum

Smell: Vertical Separation

Define, assign, and use variables and functions near where they are used.

Define local variables where first used, ideally as they are assigned.

Define private functions just below their first use. Avoid forcing the reader to scroll.

Obfuscator

Smell: Inconsistency

Be consistent in your naming, formatting, and usage patterns within your application.

Obfuscator

Smell: Poor Names

Naming things has often been cited as one of the hardest problems in computer science. Use descriptive names and avoid abbreviations and encodings where possible.

Obfuscator

Ideal Naming Characteristics

Descriptive

Appropriate Abstraction Level

Follow Standards

Unambiguous

Longer Names For Longer Scopes

Don’t Encode or Abrv8

Ideal Naming Characteristics

Descriptive

Appropriate Abstraction Level

Follow Standards

Unambiguous

Longer Names For Longer Scopes

Don’t Encode or Abbreviate

Descriptive Names

```
public static List<int> Generate(int n)
{
    var x = new List<int>();
    for (int i = 2; n > 1; i++)
        for (; n % i == 0; n /= i)
            x.Add(i);
    return x;
}
```

Descriptive Names

```
public static List<int> GeneratePrimeFactorsOf(int input)
{
    var primeFactors = new List<int>();
    for (int candidateFactor = 2; input > 1;
  candidateFactor++)
        while (input % candidateFactor == 0)
        {
            primeFactors.Add(candidateFactor);
            input /= candidateFactor;
        }
    return primeFactors;
}
```

Abstraction Level

```
private IOrderSource _orderSource; // set by constructor

public void ProcessOrder()
{
    var orderFromFile = _orderSource.GetOrder();
}
```

Abstraction Level

```
private IOrderSource _orderSource; // set by constructor

public void ProcessOrder()
{
    var order = _orderSource.GetOrder();
}
```

Follow Standards and Conventions

```
var customer = customerFactory.Create(123);
var Order = orderBuilder.Make(234);
var Orderitem = orderItemMaker.NewItem();

order.addItem(orderItem);
customer.AppendOrder(order);
```

C# Coding Conventions

https://bit.ly/2vNiniK

Unambiguous

```
Account account1 = GetAccount(accountId);
Account account2 = GetAccount(accountId2);
Account account3 = GetAccount(accountId3);

bool result = Transfer(amount, account1, account2,
account3);
```

Unambiguous

```
Account sender = GetAccount(senderAccountId);
Account recipient =
GetAccount(recipientAccountId);
Account commissionAccount =
GetAccount(commissionAccountId);

bool result = Transfer(amount, sender, recipient,
commissionAccount);
```

Long Names for Long Scopes (And Vice Versa)

```
public string ListUsers()
{
    var sb = new StringBuilder();
    for (int i = 0; i < Application.CurrentUserCount; i++)
    {
        sb.Append("User " + i + Environment.NewLine);
    }
    return sb.ToString();
}
```

Long Names for Long Scopes (And Vice Versa)

```
public string ListUsers()
{
    var sb = new StringBuilder();
    for (int i = 0; i < A.UC; i++)
    {
        sb.Append("User " + i + E.NL);
    }
    return sb.ToString();
}
```

Avoid Encodings

```
string strName;
int iCount;
DateTime dtStart;
DateTime dtEnd;
User usrOne;
User usrTwo;
SqlUserRepository surDataAccess;
List<User> lstUsers;
```

Avoid Encodings

```
string name;
int count;
DateTime StartDate;
DateTime EndDate;
User user1;
User user2;
SqlUserRepository userRepository;
List<User> users;

string userName = UserNameTextBox.Text;
UserNameLabel.Text = userName;
```

Smell: Switch Statements

Switch statements, and complex if-else chains, may indicate a lack of proper use of object-oriented design.

Object-Orientation Abuser

```
MethodOne(Class class)
{
   switch (class.TypeId)
   {
      case 1:
         case 2:
         case n:
   }
}

AnotherMethod(Class class)
{
 switch (class.TypeId)
   {
       case 1:
          case 2:
          case n:
   }
}
```

- One switch in your codebase on a particular value is probably fine

- It’s when you duplicate them that it’s a code smell.

- Why does class have a type property? Could it use inheritance to be that type?

Smell: Duplicate Code

Duplication is the root of all software evil. Follow the Don't Repeat Yourself principle and avoid repetition in your code when possible.

Disposable

Duplicate Code

```
public void Method(Customer customer, Order order, Logger logger)
{
    if(customer == null)
    {
        throw new ArgumentNullException(“Customer cannot be null”);
    }
    if(order == null)
    {
        throw new ArgumentNullException(“Order cannot be null”);
    }
    if(logger == null)
    {
        throw new ArgumentNullException(“Logger cannot be null”);
    }
    // do actual work
}
```

Duplicate Code

```
using Ardalis.GuardClauses;

public void Method(Customer customer, Order order, Logger logger)
{
    Guard.Against.Null(customer, nameof(customer));
    Guard.Against.Null(order, nameof(order));
    Guard.Against.Null(logger, nameof(logger));
    // do actual work
}
```

Duplicate Code

```
public class BasketAddItem
{
    [Fact]
    public void AddsBasketItemIfNotPresent()
    {
        var basket = new Basket();
        // test logic
    }

    [Fact]
    public void IncrementsItemQuantityIfPresent()
    {
        var basket = new Basket();
        // test logic
    }
    // a bunch more tests
}
```

Duplicate Code

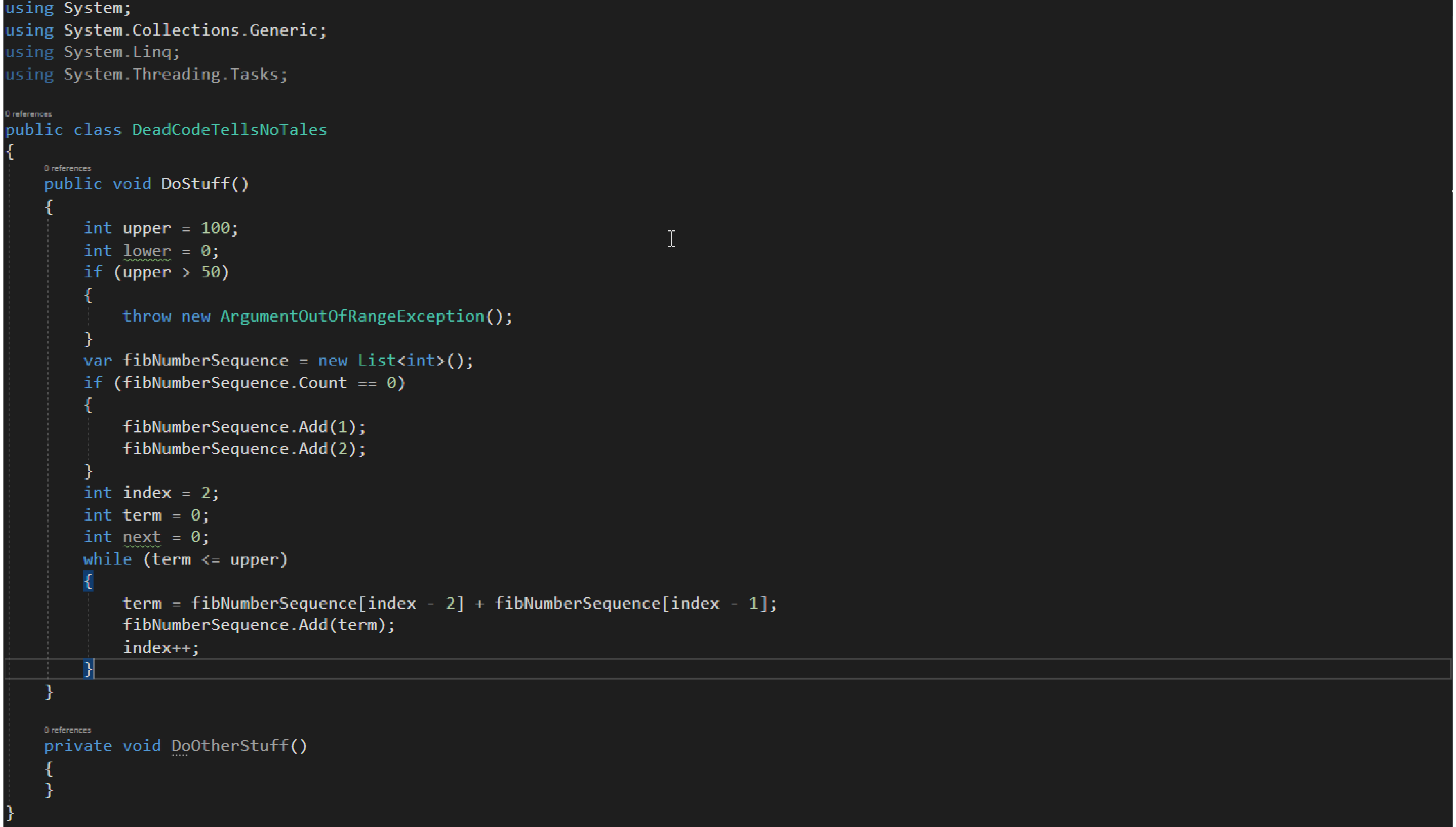
```
public class BasketAddItem
{
    private Basket _basket = new Basket();

    [Fact]
    public void AddsBasketItemIfNotPresent()
    {
        // test logic using _basket
    }
    [Fact]
    public void IncrementsItemQuantityIfPresent()
    {
        // test logic using _basket
    }
    // a bunch more tests
}
```

Smell: Dead Code

Get rid of useless code that is never executed. It’s not adding value; it’s only adding weight to the codebase. It’s a distraction. Bury it.

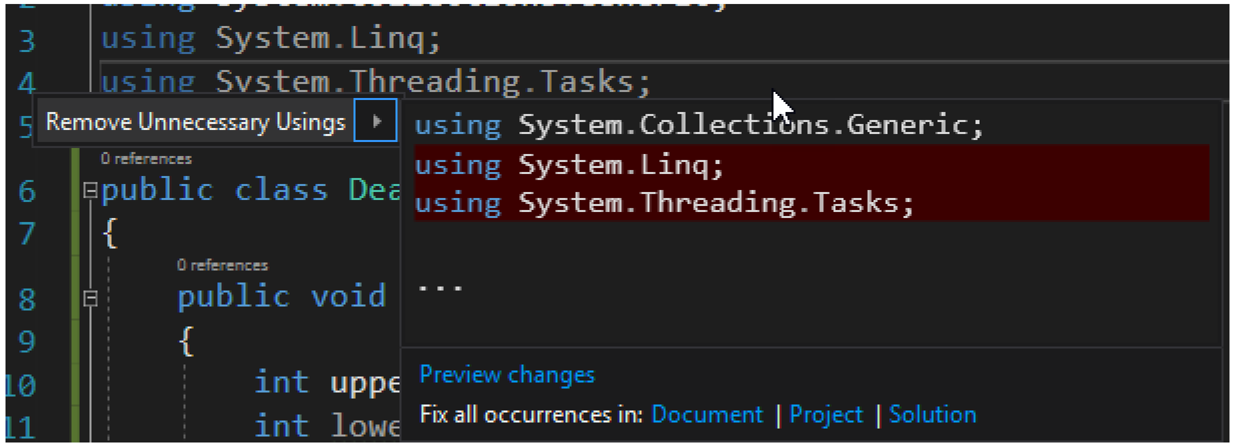
Disposable


```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

0 references
public class DeadCodeTellsNoTales
{
    0 references
    public void DoStuff()
    {
        int upper = 100;
        int lower = 0;
        if (upper > 50)
        {
            throw new ArgumentOutOfRangeException();
        }
        var fibNumberSequence = new List<int>();
        if (fibNumberSequence.Count == 0)
        {
            fibNumberSequence.Add(1);
            fibNumberSequence.Add(2);
        }
        int index = 2;
        int term = 0;
        int next = 0;
        while (term <= upper)
        {
            term = fibNumberSequence[index - 2] + fibNumberSequence[index - 1];
            fibNumberSequence.Add(term);
            index++;
        }
    }

    0 references
    private void DoOtherStuff()
    {
    }
}
```

Visual Studio Dead Code Refactorings



```
using System.Linq;
using System.Threading.Tasks;
```

Remove Unnecessary Usings

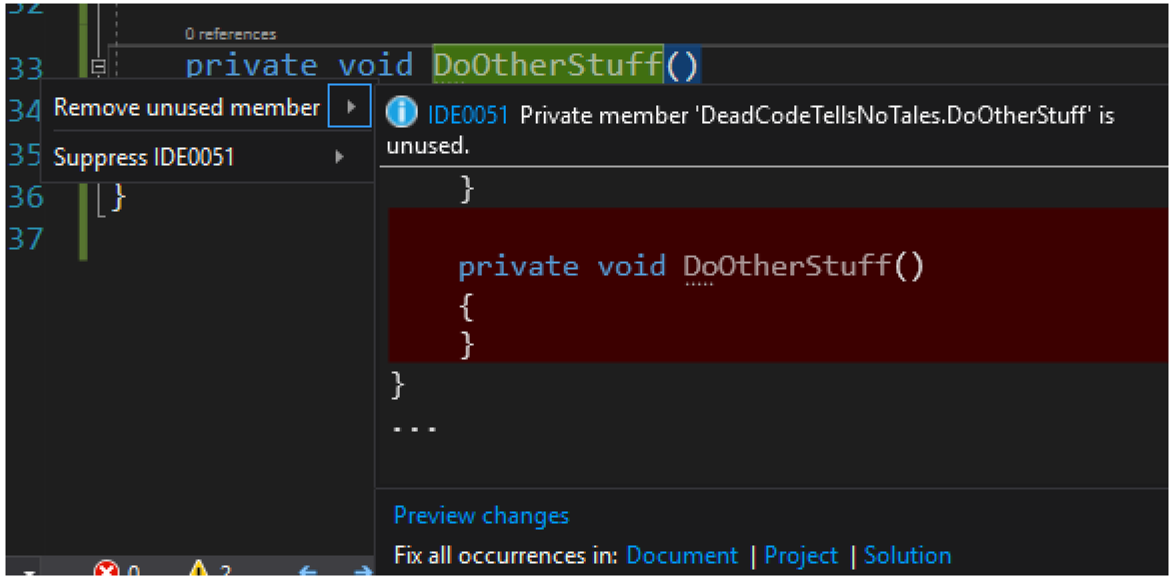
```
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

...
```

Preview changes

Fix all occurrences in: Document | Project | Solution

```
0 references
public class Dea
{
    0 references
    public void
    {
        int uppe
        int lowe
```



```
    0 references
    private void DoOtherStuff()
```

Remove unused member

Suppress IDE0051

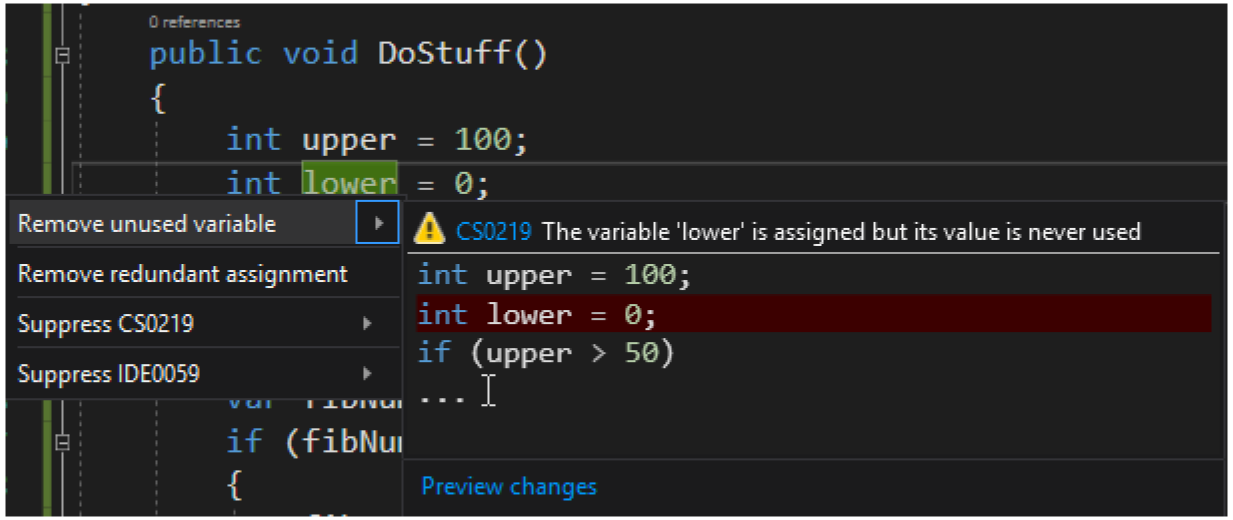
IDE0051 Private member 'DeadCodeTellsNoTales.DoOtherStuff' is unused.

```
    }

    private void DoOtherStuff()
    {
    }
}
...
```

Preview changes

Fix all occurrences in: Document | Project | Solution



```
0 references
public void DoStuff()
{
    int upper = 100;
    int lower = 0;
```

Remove unused variable

Remove redundant assignment

Suppress CS0219

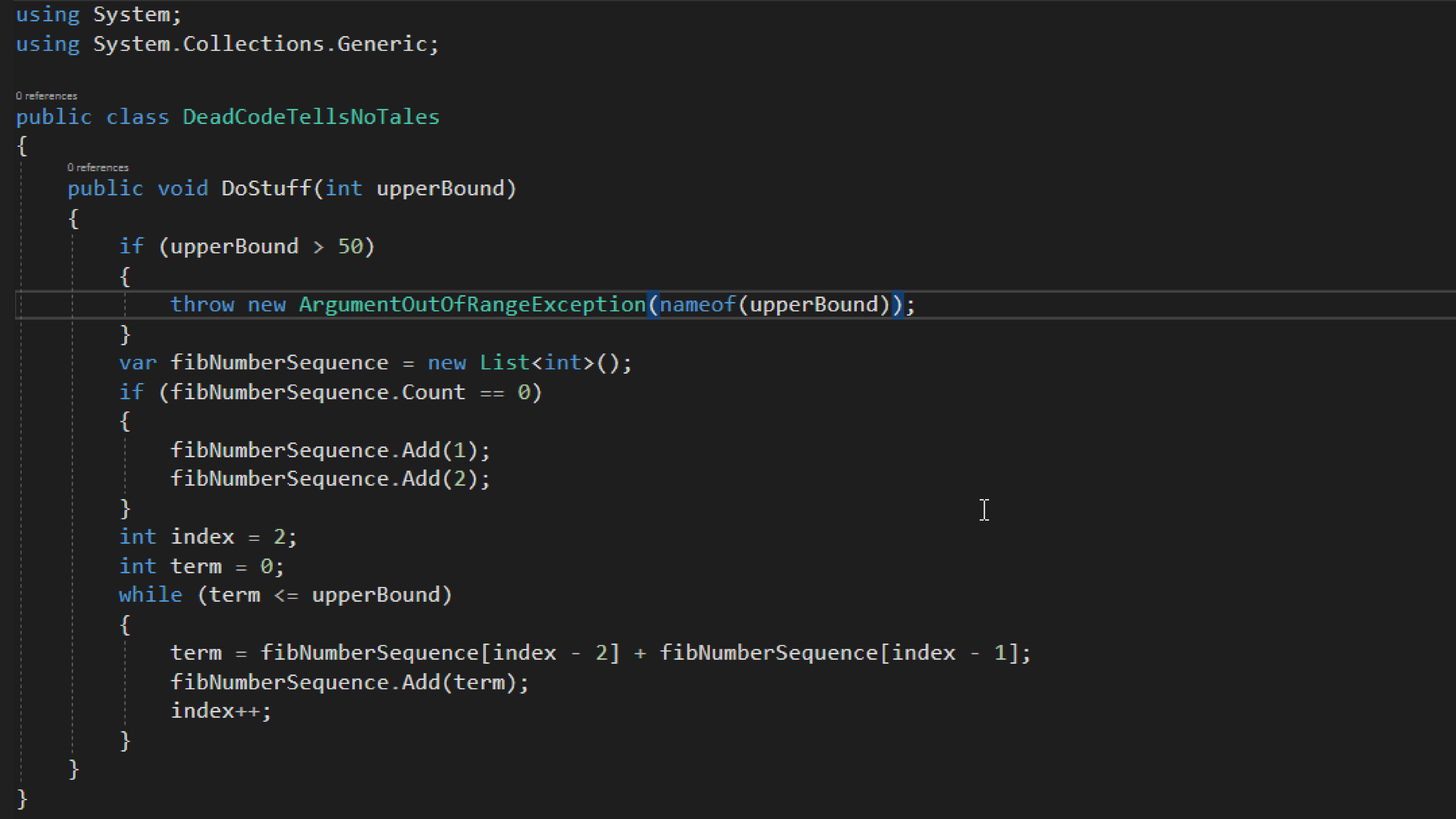
Suppress IDE0059

CS0219 The variable 'lower' is assigned but its value is never used

```
int upper = 100;
int lower = 0;
if (upper > 50)
...
```

Preview changes

```
    if (fibNu
    {
```



```csharp
using System;
using System.Collections.Generic;

public class DeadCodeTellsNoTales
{
    public void DoStuff(int upperBound)
    {
        if (upperBound > 50)
        {
            throw new ArgumentOutOfRangeException(nameof(upperBound));
        }
        var fibNumberSequence = new List<int>();
        if (fibNumberSequence.Count == 0)
        {
            fibNumberSequence.Add(1);
            fibNumberSequence.Add(2);
        }
        int index = 2;
        int term = 0;
        while (term <= upperBound)
        {
            term = fibNumberSequence[index - 2] + fibNumberSequence[index - 1];
            fibNumberSequence.Add(term);
            index++;
        }
    }
}
```

Smell: Hidden Temporal Coupling

Certain operations must be called in a certain sequence, or they won’t work. Nothing in the design forces this behavior – developers just have to figure it out from context or tribal knowledge.

Coupler

Temporal Coupling

```
PrepareCrust();

AddToppings();

Bake();

CutIntoSlices();
```

Temporal Coupling

```
public abstract class BakedGoodBase
{
    public void MakeBakedGood()
    {
        PrepareCrust();
        AddToppings();
        Bake();
        CutIntoSlices();
    }
 protected abstract void PrepareCrust();
 protected abstract void AddToppings();
 protected abstract void Bake();
 protected abstract void CutIntoSlices();
}
```

Temporal Coupling

```
public abstract class BakedGoodBase
{
    public void MakeBakedGood()
    {
        PrepareCrust();
        AddToppings();
        Bake();
        CutIntoSlices();
    }
 protected abstract void PrepareCrust();
 protected abstract void AddToppings();
 protected abstract void Bake();
 protected abstract void CutIntoSlices();
}
```

Temporal Coupling

```
Crust crust = PrepareCrust();
ToppedCrust toppedCrust = AddToppings(crust);
BakedItem bakedItem = Bake(toppedCrust);
SlicedItem slicedItem = CutIntoSlices(bakedItem);
```

Key Takeaways

Primitive Obsession

Vertical Separation

Inconsistency

Poor Names

Switch Statements

Duplicate Code

Dead Code

Hidden Temporal Coupling