Method Code Smells and Refactorings

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

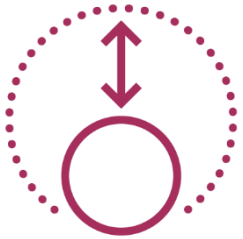
Objectives

Learn various method-related code smells

Learn refactoring techniques to address them

Smell: Long Method

Prefer shorter methods to longer methods. Small methods can have better names, because they're doing less, and since they can be well-named, are small, and don't do much, they're easier than long methods to understand.



Bloater

```
   4  namespace CodeSmells.Bloaters.LongMethod
   5  {
   6      public class ManagerUtility
   7      {
   8          public void DoEverythingForTheApplication()...
3176      }
```

How small is “small”?

Methods should fit on one screen (no scrolling)

Ideally fewer than 10 lines of code

Refactoring Long Methods

- Extract Method
- Compose Method
- Replace Nested Conditional with Guard Clause
- Replace Conditional Dispatcher with Command
- Move Accumulation to Visitor
- Replace Conditional Logic with Strategy

Visual Studio Extract Method

```
93      public IServiceProvider ConfigureServices(IServiceCollection services)
94      {
95          _logger.LogInformation("Entering ConfigureServices");
96          services.Configure<AppSettings>(Configuration);
97          services.AddAntiforgery(options => options.HeaderName = "X-XSRF-TOKEN");
98          services.AddMemoryCache();
99          services.AddSession();
100
101         services.Configure<CookiePolicyOptions>(options =>...);
107
108         services.AddIdentity<ApplicationUser, IdentityRole>(options =>...)
117             .AddEntityFrameworkStores<AppDbContext>()
118             .AddDefaultTokenProviders();
119
120         services.ConfigureApplicationCookie(options =>...);
```

```
93      public IServiceProvider ConfigureServices(IServiceCollection services)
94      {
95          services.AddSession();
96          NewMethod(services);
97
98          services.AddMvc()
99      ...
100
101     private static void NewMethod(IServiceCollection services)
107     {
108         services.Configure<CookiePolicyOptions>(options =>
117         {
118             // This lambda determines whether user consent for non-essential cookies is needed for a given request.
119             options.CheckConsentNeeded = context => true;
120             options.MinimumSameSitePolicy = SameSiteMode.None;
133         });
134
135         services.AddIdentity<ApplicationUser, IdentityRole>(options =>
136         {
137             options.SignIn.RequireConfirmedEmail = false;
138             options.Password.RequiredLength = 8;
139             options.Password.RequireLowercase = true;
140             options.Password.RequireUppercase = true;
141             options.Password.RequireNonAlphanumeric = true;
142             options.User.RequireUniqueEmail = true;
143         })
144             .AddEntityFrameworkStores<AppDbContext>()
145             .AddDefaultTokenProviders();
146
147         services.ConfigureApplicationCookie(options =>
148         {
149             options.Events.OnRedirectToLogin = context =>
150             {
151                 context.Response.StatusCode = 401;
152                 return Task.CompletedTask;
153             };
154             options.Events.OnRedirectToAccessDenied = context =>
155             {
156                 context.Response.StatusCode = 403;
157                 return Task.CompletedTask;
158             };
159         });
160
161         services.AddMvc()
162     ...
163     }
164 }
165
166
167
168
```

Extract Method

Use discard '_'

Visual Studio Extract Method

```
93      public IServiceProvider ConfigureServices(IServiceCollection services)
94      {
95          _logger.LogInformation("Entering ConfigureServices");
96          services.Configure<AppSettings>(Configuration);
97          services.AddAntiforgery(options => options.HeaderName = "X-XSRF-TOKEN");
98          services.AddMemoryCache();
99          services.AddSession();
100         NewMethod(services);
101
```

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

Guard Clauses

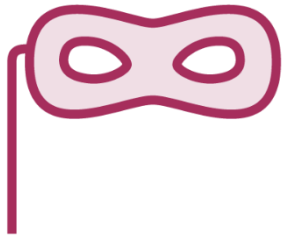
```
public void Method(Customer customer, Order order, Logger logger)
{
    if(customer == null)
    {
        throw new ArgumentNullException("Customer cannot be null");
    }
    if(order == null)
    {
        throw new ArgumentNullException("Order cannot be null");
    }
    if(logger == null)
 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

Guard Clauses

```
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
}
```

Smell: Obscured Intent

Small and dense are not ends in and of themselves! Take the time to ensure your code is intention-revealing, not obscuring.



Obfuscator

Intention Obscuring

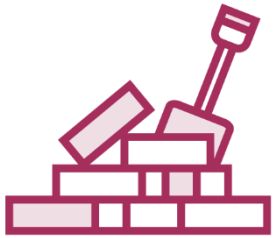
```
public int m_otCalc()
{
    return iThsWkd * iThsRte + (int)Math.Round(0.5 * iThsRte *
        Math.Max(0, iThsWkd - 400));
}
```

Intention Revealing

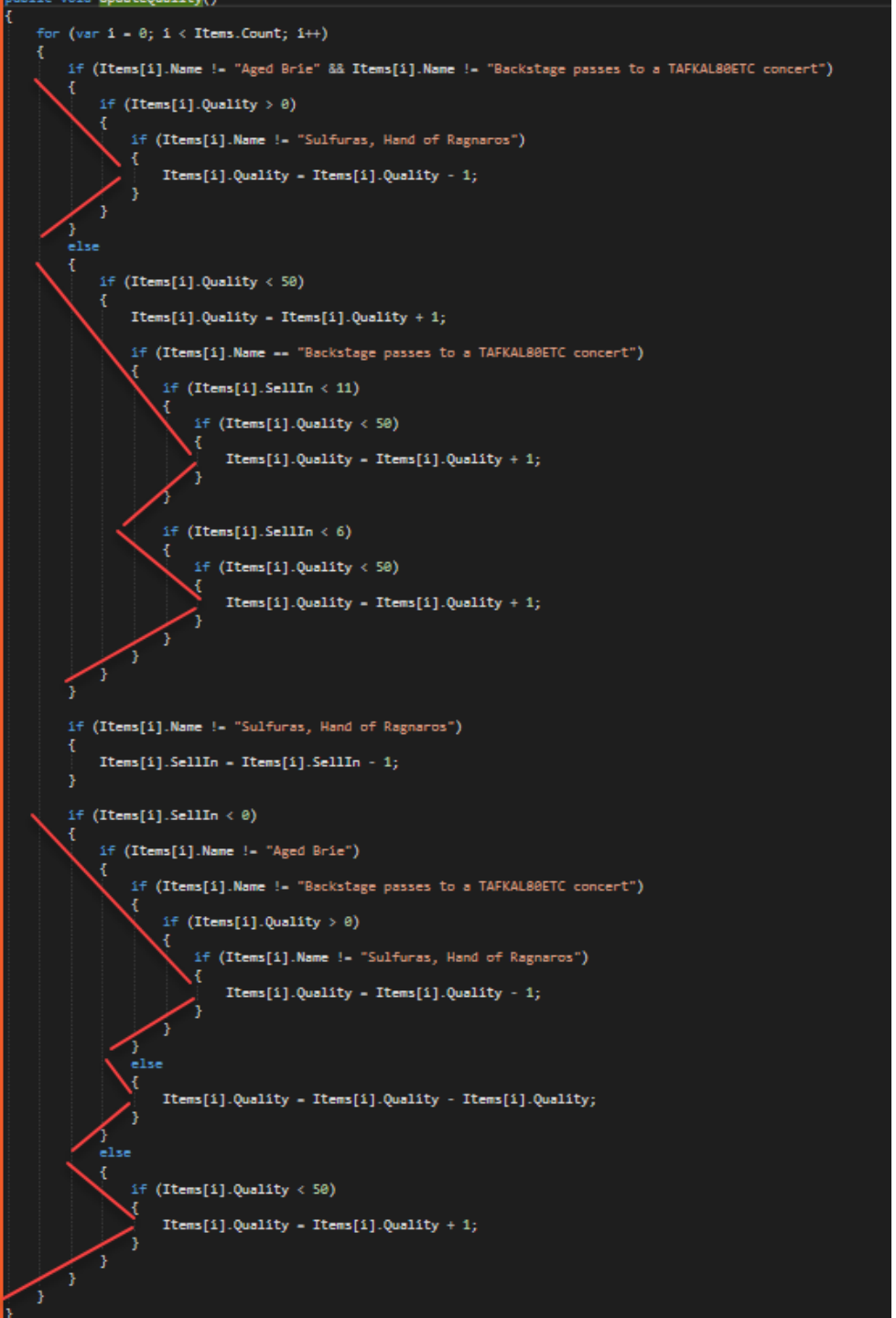
```
public int CalculateStraightPay()
{
    return tenthsWorked * tenthsRate;
}

public int CalculateOverTimePay()
{
    int overTimeTenths = Math.Max(0, tenthsWorked - 400);
    int overTimePay = CalculateOverTimeBonus(overTimeTenths);
    return CalculateStraightPay() + overTimePay;
}

public int CalculateOverTimeBonus(int overTimeTenths)
{
    double bonus = 0.5 * tenthsRate * overTimeTenths;
    return (int)Math.Round(bonus);
}
```

Smell: Conditional Complexity

Methods should limit the amount of conditional complexity they contain. The number of unique logical paths through a method can be measured as Cyclomatic Complexity, which should be kept under 10.



Change Preventer



```
{
    for (var i = 0; i < Items.Count; i++)
    {
        if (Items[i].Name != "Aged Brie" && Items[i].Name != "Backstage passes to a TAFKAL80ETC concert")
        {
            if (Items[i].Quality > 0)
            {
                if (Items[i].Name != "Sulfuras, Hand of Ragnaros")
                {
                    Items[i].Quality = Items[i].Quality - 1;
                }
            }
        }
        else
        {
            if (Items[i].Quality < 50)
            {
                Items[i].Quality = Items[i].Quality + 1;

                if (Items[i].Name == "Backstage passes to a TAFKAL80ETC concert")
                {
                    if (Items[i].SellIn < 11)
                    {
                        if (Items[i].Quality < 50)
                        {
                            Items[i].Quality = Items[i].Quality + 1;
                        }
                    }

                    if (Items[i].SellIn < 6)
                    {
                        if (Items[i].Quality < 50)
                        {
                            Items[i].Quality = Items[i].Quality + 1;
                        }
                    }
                }
            }
        }

        if (Items[i].Name != "Sulfuras, Hand of Ragnaros")
        {
            Items[i].SellIn = Items[i].SellIn - 1;
        }

        if (Items[i].SellIn < 0)
        {
            if (Items[i].Name != "Aged Brie")
            {
                if (Items[i].Name != "Backstage passes to a TAFKAL80ETC concert")
                {
                    if (Items[i].Quality > 0)
                    {
                        if (Items[i].Name != "Sulfuras, Hand of Ragnaros")
                        {
                            Items[i].Quality = Items[i].Quality - 1;
                        }
                    }
                }
                else
                {
                    Items[i].Quality = Items[i].Quality - Items[i].Quality;
                }
            }
            else
            {
                if (Items[i].Quality < 50)
                {
                    Items[i].Quality = Items[i].Quality + 1;
                }
            }
        }
    }
}
```

Indented conditionals form a “mountain range”

Conditional logic obscures actual statements being executed

Visual Studio Code Metrics

Code Metrics Results

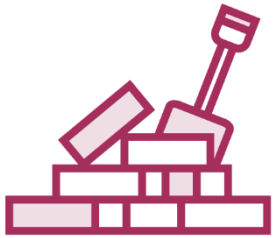
Filter: None Min: Max:

Hierarchy	Maintainability In...	Cyclomatic Com...	Depth of Inherita...	Class Coupling	Lines of Code
RefactoringSamples (Debug)	89	39	3	14	265
RefactoringSamples.MethodLevel.ConditionalComplexity	76	26	1	5	125
GildedRoseConsoleProgram	52	20	1	5	112
Items : IList<Item>	100	0		2	1
Main(string[]) : void	68	1		5	28
UpdateQuality() : void	45	19		2	76
Item	100	6	1	0	9
RefactoringSamples.StatementLevel	93	13	3	9	140

https://ardalis.com/measuring-aggregate-complexity-in-software-applications

Smell: Inconsistent Abstraction Level

Methods should operate at a consistent abstraction level. Don't mix high level and low level behavior in the same method.



Change Preventer

Specific Method Refactorings

Extract Method

1. **Identify the code to extract**
2. **Create a new method with a good name**
3. **Copy the code from the source to the new method**
   - Temporary variables used only in this code can be copied over as well
4. **Identify modified local variables**
   - Does the extracted method need to return a variable?
   - If more than one, consider extracting a smaller method.
5. **Compile**
6. **Replace extracted code with call to new method**
7. **Compile and Run Tests**

Rename Method

1. Is the method implemented by sub-/super class?
   - If so, repeat these steps for each implementation
2. Create a new method with the same parameters and the new name
3. Copy (don't cut) the old method body into the new one.
4. Compile
5. Change the old method to call the new method (optional)
6. Compile and Run Tests
7. Find all references to old method and update to new
8. Remove (or mark [Obsolete]) the old method
9. Compile and Run Tests

Inline Method

1. Confirm no subclasses override the method
2. Find all calls to the method
3. Replace each call with the method's body
4. Compile and Run Tests
5. Remove the original method
6. Compile and Run Tests

Inline Method

```
public void UpdateQuality()
{
    if(NameContainsBackstagePasses(Items[i].Name))
    {
        // do something
    }
}

public bool NameContainsBackstagePasses(string name)
{
    return name.Contains(“Backstage passes”);
}
```

Inline Method

```
public void UpdateQuality()
{
    if(Items[i].Name.Contains(“Backstage passes”))
    {
        // do something
    }
}
```

Introduce Explaining Variable

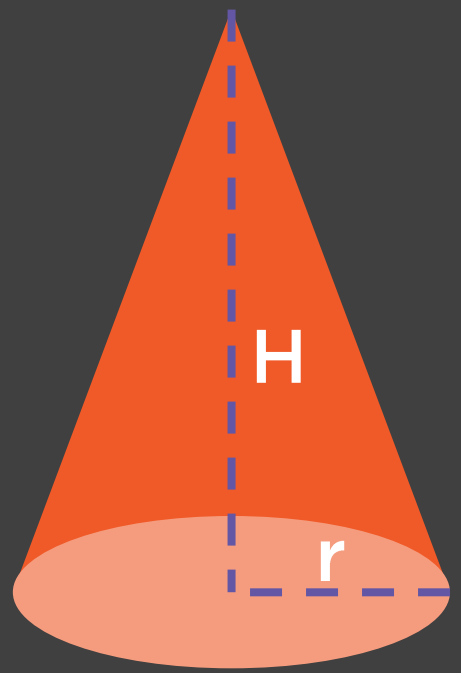
1. **Declare a temp variable and set it to part of the complex expression**
   - Make sure it has a clear name
2. **Replace the result part of the expression with the temp**
3. **Compile and Run Tests**

Repeat as needed for other parts of the expression

Introduce Explaining Variable

$$V = \pi r^2 H/3$$
$$A = \pi r^2$$

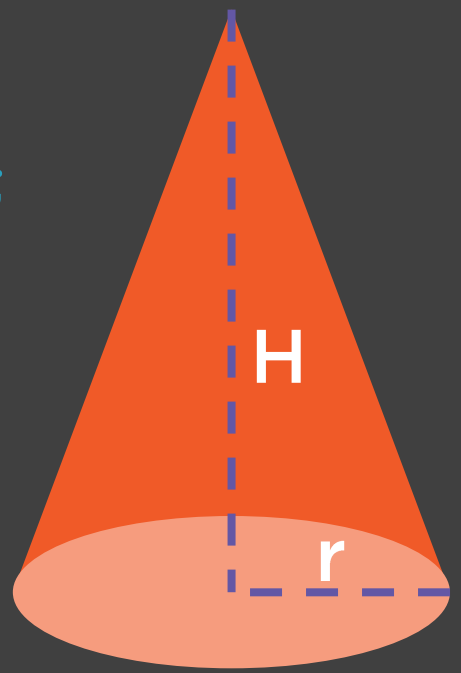
```
public decimal CalculateConeVolume()
{
    return Math.Pi * radius * radius * height / 3;
}
```



Introduce Explaining Variable

$$V = \pi r^2 H/3$$
$$A = \pi r^2$$

```
public decimal CalculateConeVolume()
{
    decimal coneOpeningArea = Math.Pi * radius * radius;
    return coneOpeningArea * height / 3;
}
```



Inline Temp

1. Confirm the temp variable is only assigned once
2. Replace references to the temp with the right side of the temp's assignment operation
3. Compile and Run Tests (each time)
4. Remove the temp declaration and assignment
5. Compile and Run Tests

Replace Temp With Query

1. Confirm the temp variable is only assigned once
2. Extract the right-hand side of the assignment to a method
3. Compile and Run Tests
4. Replace references to the temp with calls to the new method
5. Compile and Run Tests (each time)
6. Remove the temp declaration and assignment
7. Compile and Run Tests

Split Temporary Variable

1. Change the name of the temp at its declaration and first assignment
2. Confirm the new temp is not assigned elsewhere
3. Change all references of the temp up to its second assignment to use the new name
4. Declare the original temp at its second assignment
5. Compile and Run Tests

Repeat until each temp is only assigned where declared.

Parameterize Methods

1. Create a new parameterized method that can be substituted for each repetitive method
2. Compile
3. Replace all calls to one old method with a call to the new method
4. Compile and Run Tests
5. Repeat for each method, testing each time
6. Remove the original methods if they are no longer used
   - Optional: replace their bodies with calls to the parameterized method
7. Compile and Run Tests

Replace Parameter with Explicit Methods

1. Create a separate, explicit method for each value of the parameter in the original method
2. Move condition body logic to explicit methods and replace condition bodies with calls to new methods
3. Compile and Run Tests (after each change)
4. Replace each call to the original parameterized method with a call to the appropriate new method
5. Compile and Run Tests (after each change)
6. Remove the parameterized method.
7. Compile and Run Tests

Replace Parameter with Explicit Methods

```
public void UpdateValue(string property, int value)
{
    if(property==“height”)
    {
        _height = value;
    }
    if(property==“width”)
    {
        _width = value;
    }
    throw new InvalidOperationException();
}
```

Replace Parameter with Explicit Methods

```
public void UpdateHeight(int value)
{
    _height = value;
}

public void UpdateWidth(int value)
{
    _width = value;
}
```

Add Parameter

1. Check if the signature is also in a sub or superclass
   - Repeat these steps for each one
2. Declare a new method with the new parameter(s)
3. Copy the old method body into the new method
4. Compile
5. Change the old method to call the new one
6. Compile and Run Tests
7. Find all references to the old method and change to use the new one
8. Compile and Run Tests (after each change)
9. Remove the original method
10. Compile and Run Tests

Remove Parameter

1. Check if the signature is also in a sub or superclass
   - If so, DO NOT PERFORM THIS REFACTORING
2. Declare a new method without the parameter
3. Copy the old method body into the new method
4. Compile
5. Change the old method to call the new one
6. Compile and Run Tests
7. Find all references to the old method and change to use the new one
8. Compile and Run Tests (after each change)
9. Remove the original method
10. Compile and Run Tests

Separate Query from Modifier

1. **Create a query method that returns the same value as the original method**
2. **Modify the original method to return the result of a call to the new query method**
3. **Compile and Run Tests**
4. **Find all references to the old method and change them:**
   - Call the original method (without assigning it)
   - Call the query method (and assign its return value)
5. **Compile and Run Tests (after each change)**
6. **Update the original method to return void and remove all return statements from it**
7. **Compile and Run Tests**

Separate Query from Modifier

```
public IEnumerable<Invoice> ProcessOverdueInvoices(DateTime processDate)
{
    foreach (var invoice in Invoices.Where(i => (!i.Paid &&
i.PaymentDueDate < processDate)))
    {
        if (Status != AccountStatus.Overdue)
        {
            UpdateStatus(AccountStatus.Overdue);
        }
        SendPastDueNotice(invoice);
    }
    return Invoices.Where(i => (!i.Paid && i.PaymentDueDate <
processDate));
}
```

Separate Query from Modifier

```
public void ProcessAccounts(IEnumerable<Account> accounts)
{
    foreach (var account in accounts)
    {
        var overdueInvoices = account.ProcessOverdueInvoices(DateTime.Now);

        UpdateReport(overdueInvoices);
    }
}
```

Separate Query from Modifier

```
public IEnumerable<Invoice> ListPastDueInvoices(DateTime processDate)
{
    return Invoices.Where(i => (!i.Paid && i.PaymentDueDate < processDate));
}

public IEnumerable<Invoice> ProcessOverdueInvoices(DateTime processDate)
{
    foreach (var invoice in Invoices.Where(i => (!i.Paid && i.PaymentDueDate <
processDate)))
    {
        if (Status != AccountStatus.Overdue)
        {
            UpdateStatus(AccountStatus.Overdue);
        }
        SendPastDueNotice(invoice);
    }
    return ListPastDueInvoices(processDate);
}
```

Separate Query from Modifier

```
public void ProcessAccounts(IEnumerable<Account> accounts)
{
    foreach (var account in accounts)
    {
        var overdueInvoices = account.ProcessOverdueInvoices(DateTime.Now);

        UpdateReport(overdueInvoices);
    }
}
```

Separate Query from Modifier

```
public void ProcessAccounts(IEnumerable<Account> accounts)
{
    foreach (var account in accounts)
    {
        account.ProcessOverdueInvoices(DateTime.Now);
        var overdueInvoices = account.ListPastDueInvoices(DateTime.Now);

        UpdateReport(overdueInvoices);
    }
}
```

Separate Query from Modifier

```
public void ProcessAccounts(IEnumerable<Account> accounts)
{
    foreach (var account in accounts)
    {
        DateTime processDate = DateTime.Now;
        account.ProcessOverdueInvoices(processDate);
        var overdueInvoices = account.ListPastDueInvoices(processDate);

        UpdateReport(overdueInvoices);
    }
}
```

Separate Query from Modifier

```
public IEnumerable<Invoice> ListPastDueInvoices(DateTime processDate)
{
    return Invoices.Where(i => (!i.Paid && i.PaymentDueDate < processDate));
}

public IEnumerable<Invoice> ProcessOverdueInvoices(DateTime processDate)
{
    foreach (var invoice in Invoices.Where(i => (!i.Paid && i.PaymentDueDate <
processDate)))
    {
        if (Status != AccountStatus.Overdue)
        {
            UpdateStatus(AccountStatus.Overdue);
        }
        SendPastDueNotice(invoice);
    }
    return ListPastDueInvoices(processDate);
}
```

Separate Query from Modifier

```
public IEnumerable<Invoice> ListPastDueInvoices(DateTime processDate)
{
    return Invoices.Where(i => (!i.Paid && i.PaymentDueDate < processDate));
}

public void ProcessOverdueInvoices(DateTime processDate)
{
    foreach (var invoice in Invoices.Where(i => (!i.Paid && i.PaymentDueDate <
processDate)))
    {
        if (Status != AccountStatus.Overdue)
        {
            UpdateStatus(AccountStatus.Overdue);
        }
        SendPastDueNotice(invoice);
    }
}
```

Separate Query from Modifier

```
public IEnumerable<Invoice> ListPastDueInvoices(DateTime processDate)
{
    return Invoices.Where(i => (!i.Paid && i.PaymentDueDate < processDate));
}

public void ProcessOverdueInvoices(DateTime processDate)
{
    foreach (var invoice in Invoices.Where(i => (!i.Paid && i.PaymentDueDate <
processDate)))
    {
        if (Status != AccountStatus.Overdue)
        {
            UpdateStatus(AccountStatus.Overdue);
        }
        SendPastDueNotice(invoice);
    }
}
```

Separate Query from Modifier

```
public IEnumerable<Invoice> ListPastDueInvoices(DateTime processDate)
{
    return Invoices.Where(i => (!i.Paid && i.PaymentDueDate < processDate));
}

public void ProcessOverdueInvoices(DateTime processDate)
{
    foreach (var invoice in ListPastDueInvoices(processDate))
    {
        if (Status != AccountStatus.Overdue)
        {
            UpdateStatus(AccountStatus.Overdue);
        }
        SendPastDueNotice(invoice);
    }
}
```

Key Takeaways

- Long Method
- Obscured Intent
- Conditional Complexity
- Inconsistent Abstraction Level
- Extract Method <-> Inline Method
- Rename Method
- Introduce Explaining Variable <-> Inline Temp
- Replace Temp with Query
- Split Temporary Variable
- Parameterize Methods <-> Replace Parameter with Explicit Methods
- Add Parameter <-> Remove Parameter
- Separate Query from Modifier