Refactoring to Clean Code

Xavier Morera

Helping .NET developers create amazing applications

@xmorera / www.xaviermorera.com / www.bigdatainc.org

Refactoring Operations and Techniques

Extracting methods

Inline methods

Extracting variables

Parameter assignments

Refactoring Operations and Techniques

Replacing arrays with objects

Field encapsulation

Conditional expressions

Rename

Refactoring Operations and Techniques

Extracting methods

Inline methods

Extracting variables

Parameter assignments

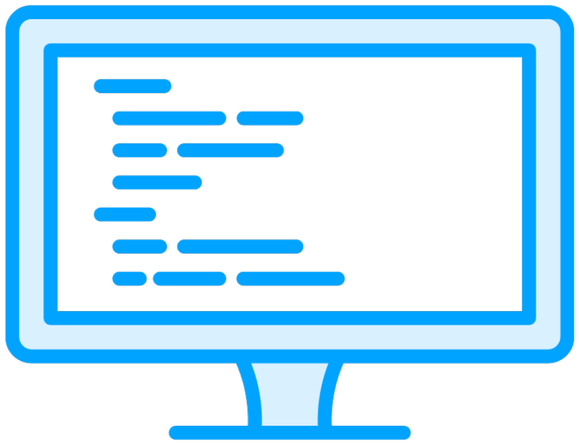
Field encapsulation

Conditional expressions

Rename

Composing Methods

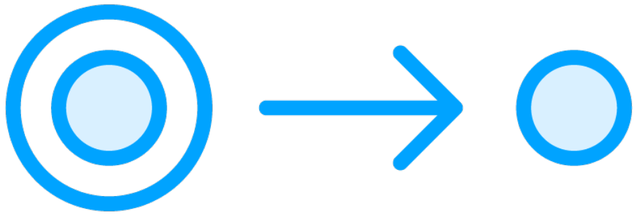
Methods



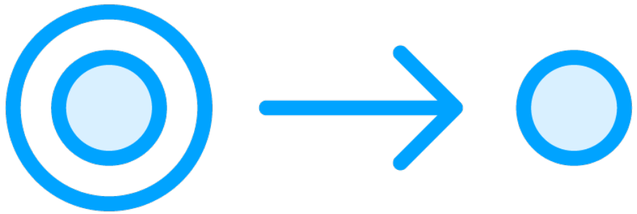
Methods must be

- Concise
- Easy to understand
- Maintainable
- Upgradeable

If any of your methods do not meet these characteristics, they may possibly require refactoring



If any of your methods do not meet these characteristics, they may possibly require refactoring



Factoring

Splitting functions into smaller functions

Refactoring

Process of restructuring existing code without changing the external behavior

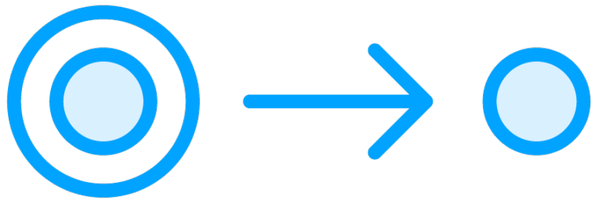
Intended to improve the design, structure, and/or implementation of the code

Refactoring

Process of restructuring existing code without changing the external behavior

Intended to improve the design, structure, and/or implementation of the code

Refactoring



Many reasons for refactoring

Excessively long methods is a common scenario

- Hard to understand underlying logic
- Even harder to change

Different refactoring techniques available

- Composing methods
  - Very useful and commonly used

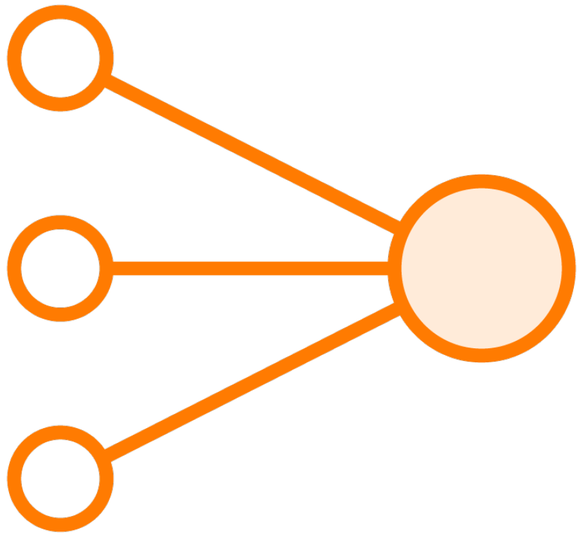
Composing Methods

Extract method

Extract Variable

Inline temp

Extract Method



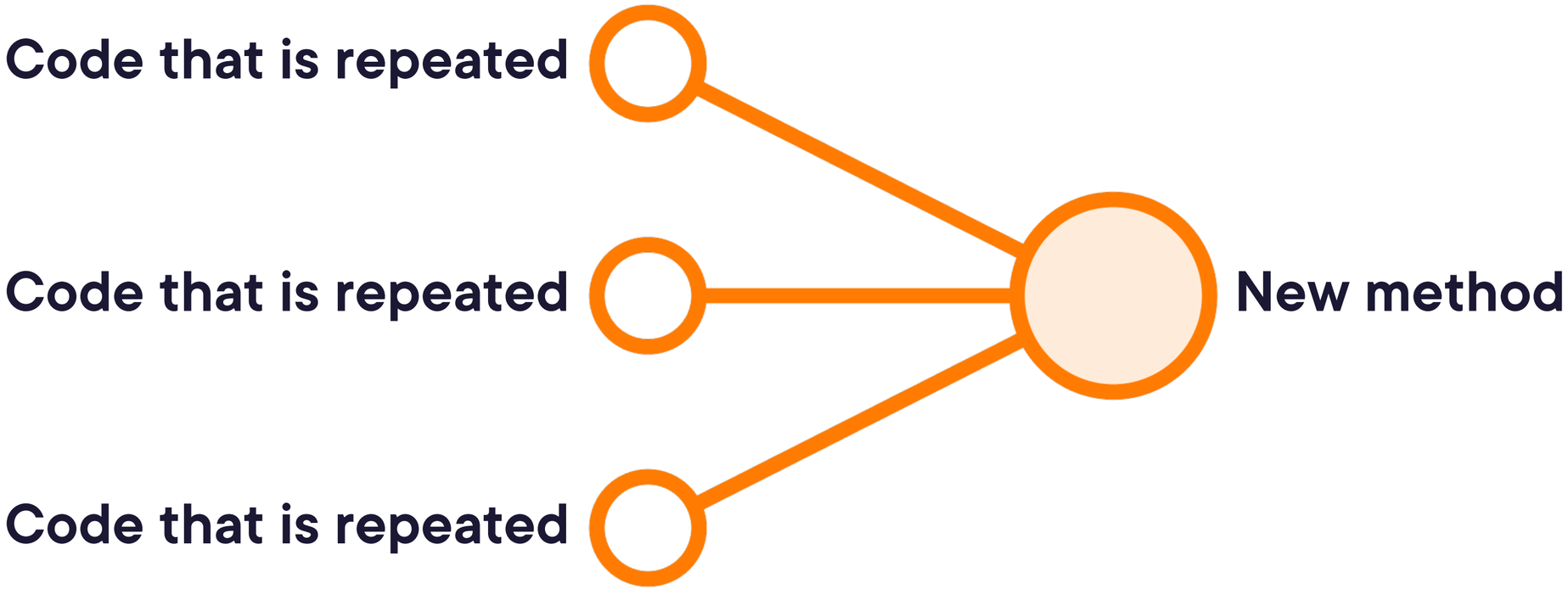
Repeated functionality

- Extract into a separate method
- DRY principle

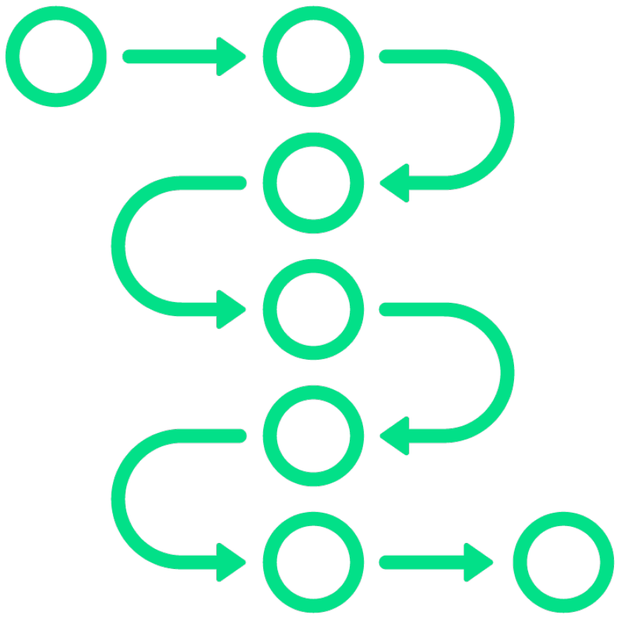
Especially when it is business-specific logic

Except if the code is very simple

Extract Method



Extract Variable



Scenario

- An expression is hard to understand

Solution

- Separate into different statements
- Each statement performs one calculation
- Stores in a temporary variable

Final result calculated from the temp variables

- Self-explanatory

Extract Variable

This expression is hard to understand

```
public double TotalPrice(Order order)
{
    return order.Quantity * order.Price - Math.Max(0, order.Quantity - 600) *
order.Price * 0.07 + Math.Min(order.Quantity * order.Price * 0.3, 100);
}
```

Extract Variable

This expression is hard to understand

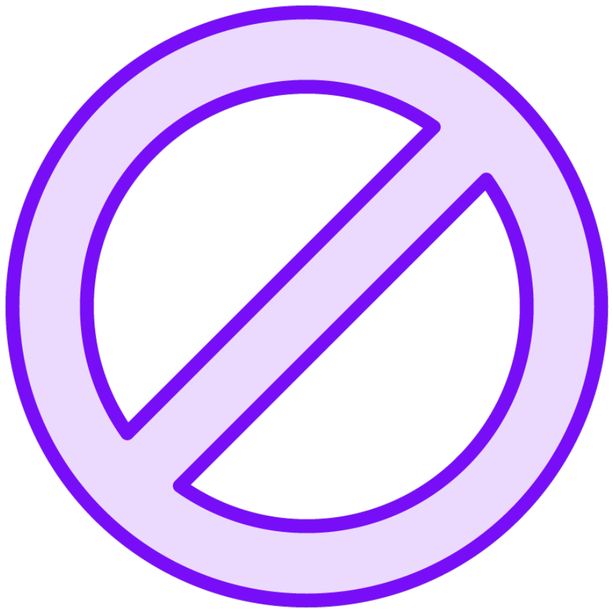
```
public double TotalPrice(Order order)
{
    return order.Quantity * order.Price - Math.Max(0, order.Quantity - 600) *
order.Price * 0.07 + Math.Min(order.Quantity * order.Price * 0.3, 100);
}
```

Extract Variable

Statements are easy to understand and self-explanatory

```
public double TotalPrice(Order order)
{
    double itemPrice = order.Price;
    int itemQuantity = order.Quantity;
    double basePrice = itemQuantity * itemPrice;
    double quantityDiscount = Math.Max(0, itemQuantity - 600) * itemPrice * 0.07;
    double shipping = Math.Min(basePrice * 0.3, 100);
    return basePrice - quantityDiscount + shipping;
}
```

Inline Temp



Scenario

- A temporary variable assigned to the result of a simple expression
- Not used for anything else

Instead, replace the reference to the variable with the expression

Inline Temp

```
public static double UpdatePrice(double total, double discountPercent)
{
    var discount = total * discountPercent;
    total -= discount;
    return total;
}
```

Inline Temp

```
public static double UpdatePrice(double total, double discountPercent)
{
    return total - (total * discountPercent);
}
```

Moving Features between Objects

Moving Features between Objects

In object-oriented programming

- Class encapsulates functionality and data

Functionality not directly related with the class

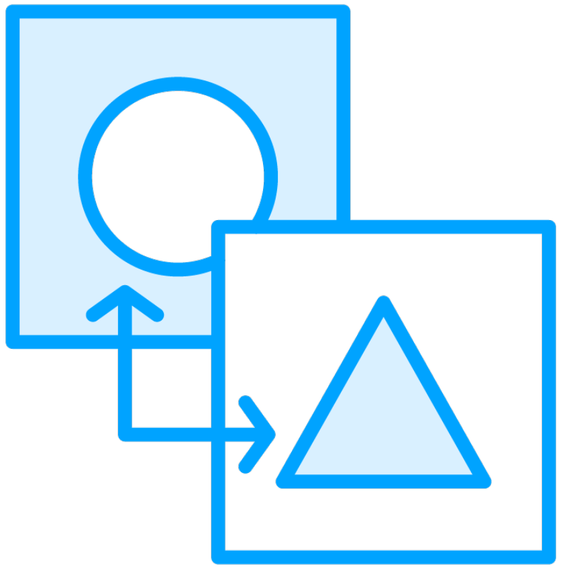
- Should be moved where it is more relevant

Moving Features between Objects

Move methods and fields

Extract class

Move Methods and Fields



Scenario

- A method or field is used in a different class
- Not where it is supposed to be

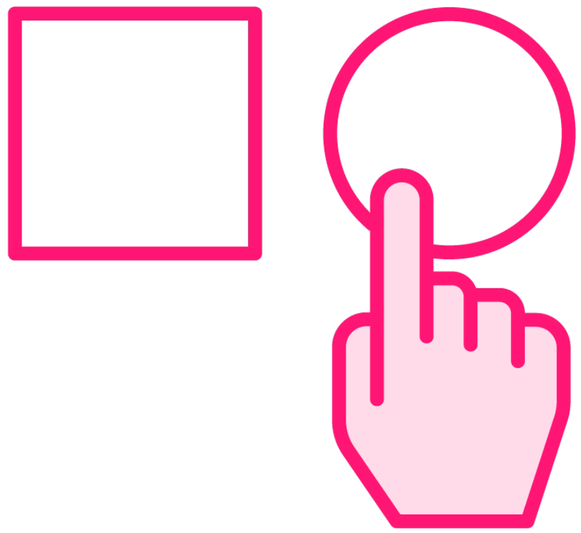
Solution

- Move to the relevant class

Demo

Move methods and fields

Extract Class



Scenario

- A class performs the functionality of two or more classes

Solution

- Create a new class
- Split the functionality accordingly
- Each class now performs its own responsibilities as intended

Except on the inline class scenario

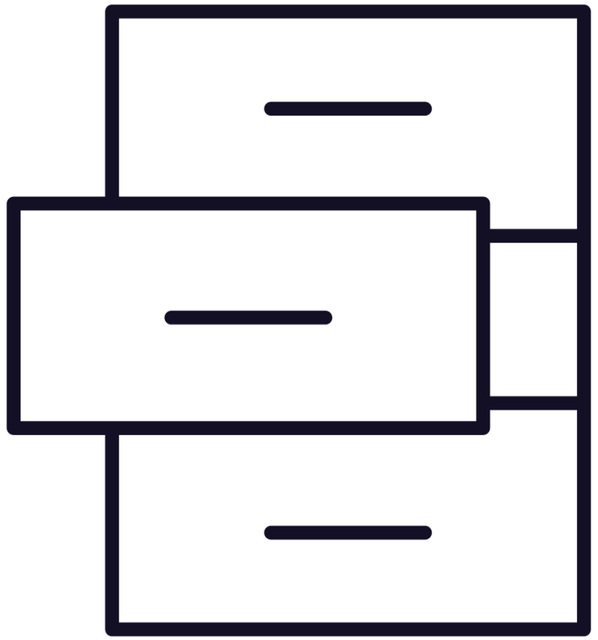
- A class does almost nothing and no future functionality in the roadmap

Demo

Extract class

Organizing Data

Organizing Data



Class associations can become complex

Properly organized and associated classes

- Can make them more portable and reusable

Techniques that help with data handling and class organization

Organizing Data

Encapsulate fields and collections

Replace strings for Enums

Replace magic numbers with symbolic constants

Change type field with class

Encapsulate Fields and Collections

Scenario

- Public field exposed in a class

Recommendation

- Create a getter and setter for the field
- Only access through the property

Demo

Encapsulate fields and collections

Replace Strings for Enums

Scenario

- A string is used to represent a set of values

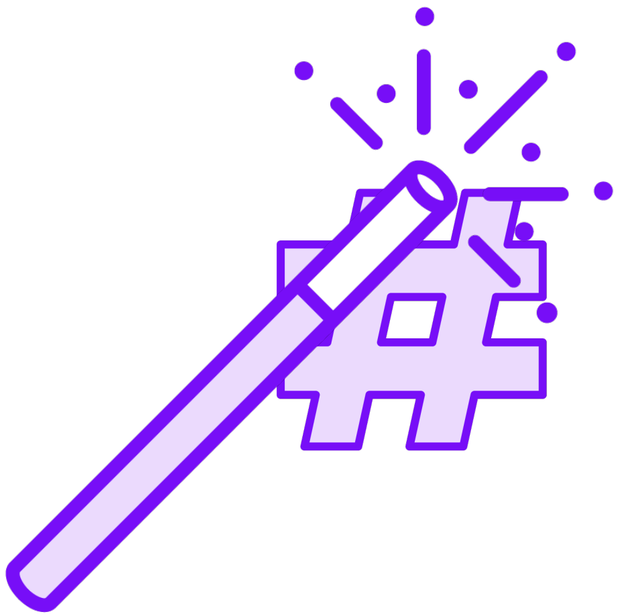
Solution

- Replace strings with Enums, structs, or objects

Demo

Replace strings for Enums

Replace Magic Numbers with Symbolic Constants



Scenario

- Common to use numbers in code that have an associated meaning
  - For example, 404 means “not found” in HTTP response codes
- Using numbers directly in code can be confusing

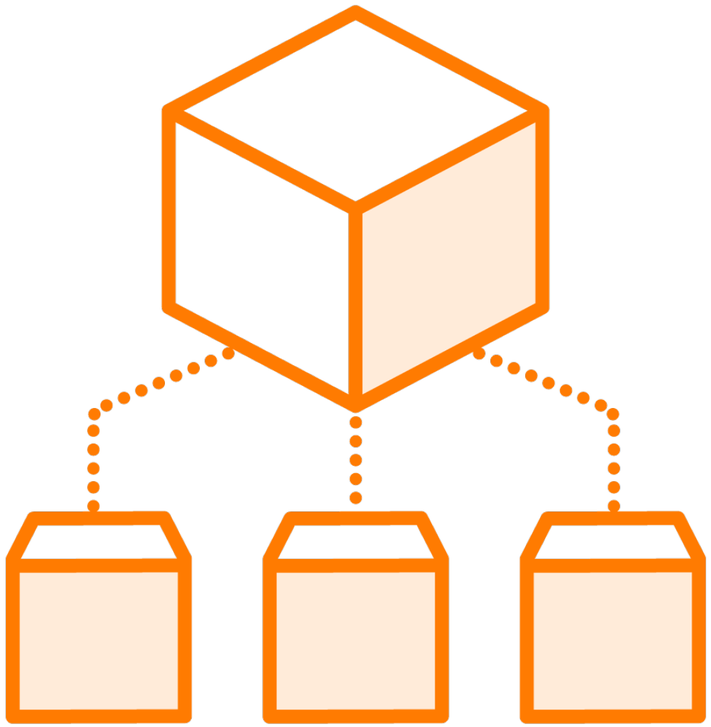
Solution

- Replace numbers with constants that have self-explanatory human-readable names

Demo

Replace magic numbers with symbolic constants

Change Type Code with Subclasses



Scenario

- String used to specify a type
- Affects behavior of the class

Solution

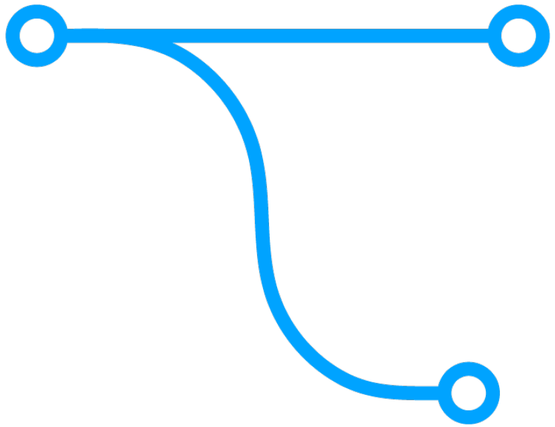
- Create subclasses for each coded type
- Extract behaviors to each individual class

Demo

Change type code with subclasses

Simplifying Conditional Expressions

Simplifying Conditional Expressions



Conditional expressions are used to control program flow logic

- Tend to get more complex over time

Simplify conditional expressions to make code more readable

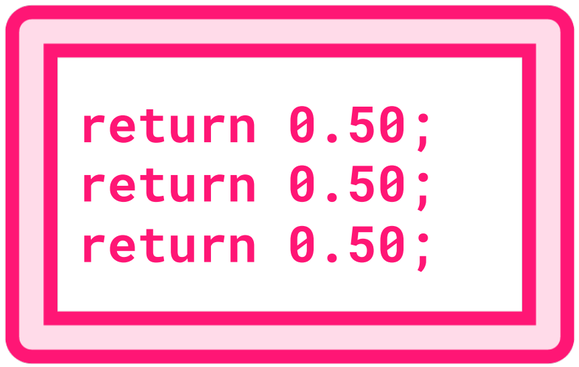
Simplifying Conditional Expressions

Consolidate conditional expression

Consolidate duplicate conditional fragments

Decompose conditional

Consolidate Conditional Expression



Scenario

- Multiple conditionals return the same value

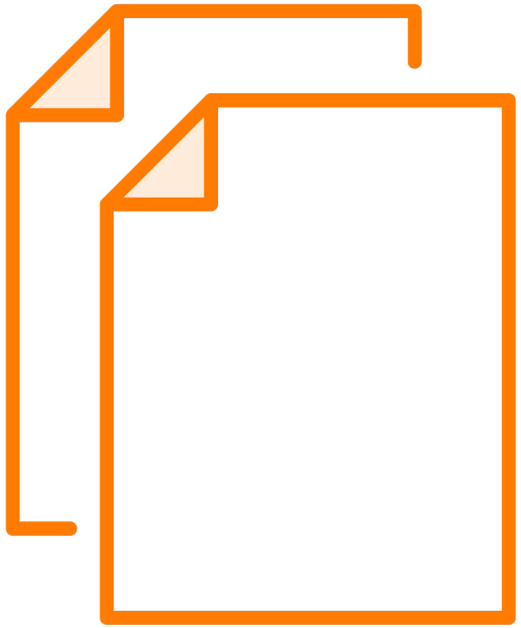
Recommendation

- Group conditions and return only once

Demo

Consolidate conditional expression

Consolidate Duplicate Conditional Fragments



Scenario

- Multiple conditionals have repeated lines

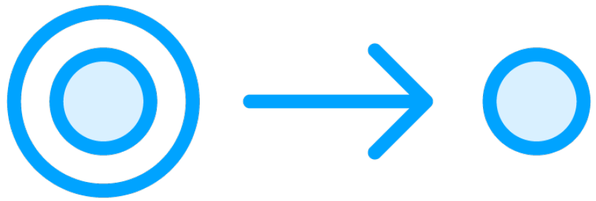
Recommendation

- Extract them
- Execute before or after the conditionals

Demo

Consolidate duplicate conditional fragments

Decompose Conditional



Scenario

- Conditional expressions are complex

Recommendation

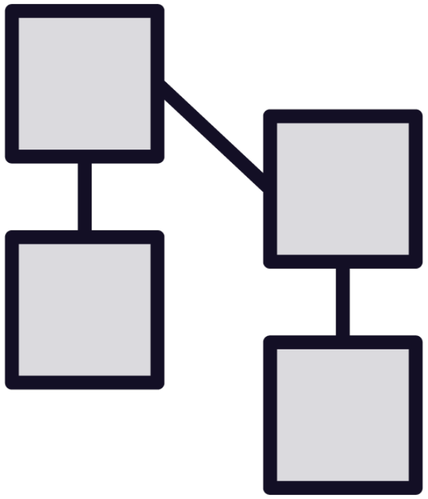
- Extract to methods to improve readability

Demo

Decompose conditional

Simplifying Method Calls

Simplifying Method Calls

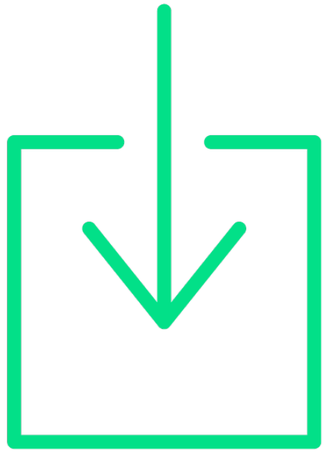


Methods make code more readable, organized, and cleaner

Multiple techniques available for simplifying method calls

- In turn, simplifies interfaces for interaction between classes

Preserve Whole Object



Scenario

- Pass multiple values of an object as parameter

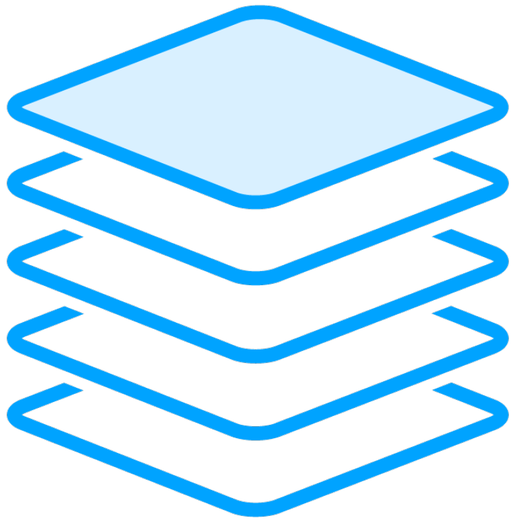
Recommendation

- Pass the entire object as parameter

Demo

Preserve whole object

Introduce Parameter Object



Scenario

- Many methods receive the same combination of parameters

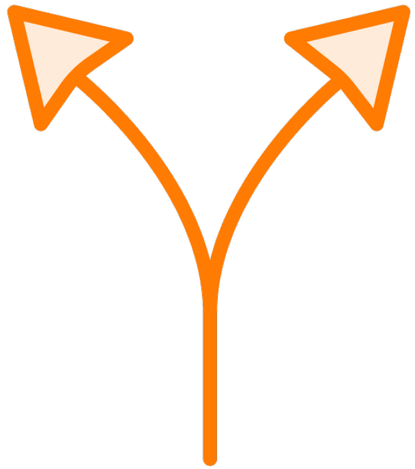
Recommendation

- Create an object that includes these parameters

Demo

Introduce parameter object

Split Methods



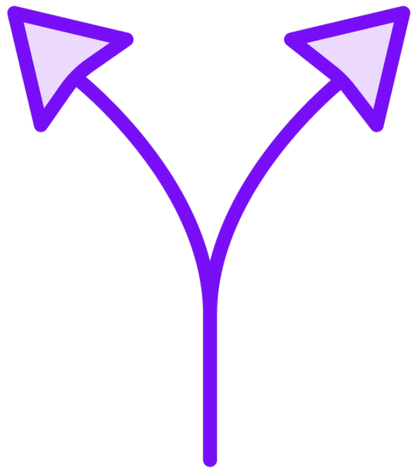
Scenario

- If a method does many things

Recommendation

- Can be split into different methods

Merge Methods



Scenario

- If several methods are exactly the same
- Except for a single line

Recommendation

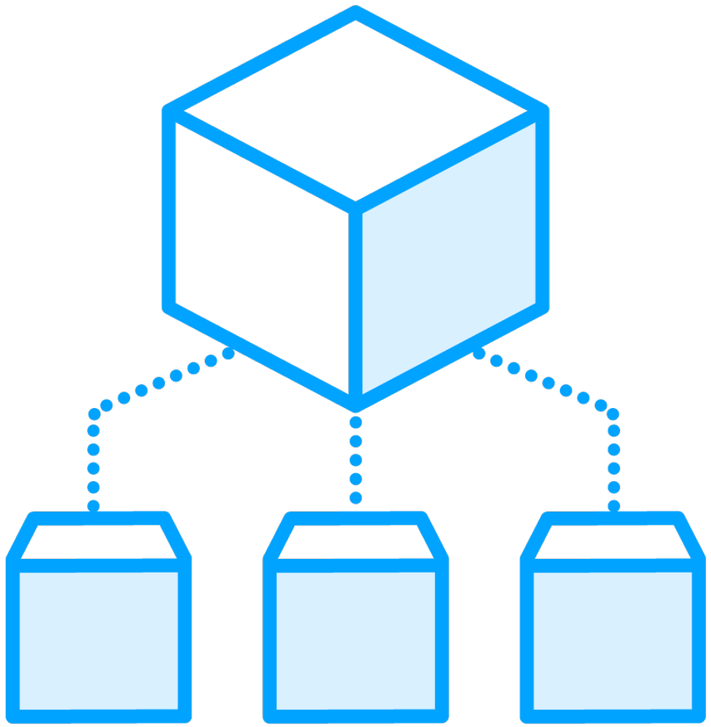
- Receive a value as parameter to conditionally execute the statement

Demo

Split and merge methods

Dealing with Generalization

Dealing with Generalization



Set of techniques that involve

- Moving functionality along the class inheritance hierarchy

Create new classes and interfaces, replacing inheritance with delegation, and vice versa

Pull Up Field

Scenario

- If two child classes have the same attribute,

Recommendation

- Move it to the parent class

Pull Up Method

Scenario

- If two child classes have the same method

Recommendation

- Move it to the parent class

Push Down Field

Scenario

- If only one child has an attribute of the parent class

Recommendation

- Move it to the child to avoid using nulls

Push Down Method

Scenario

- If only one child implements a specific method of the parent class

Recommendation

- Move it to the child to avoid
  - throw NotImplementedException

Extract Interface

Scenario

- If part of the interface in child classes is the same

Recommendation

- Move it to a separate interface

Extract Subclass

Scenario

- If a class has features that are used only in certain cases

Recommendation

- Cast these features to a subclass and use it in those cases

Extract Superclass

Scenario

- If child classes have attributes and methods in common

Recommendation

- Create a parent class and move them there

Demo

Dealing with generalization

Takeaway

Refactoring

- Process of restructuring existing code without changing the external behavior
- Intended to improve the design, structure, and/or implementation of the code

Many reasons for refactoring

- Long methods
- Non-representative names

Takeaway

Multiple techniques that can be used

- Extract method
- Extract variable
- Inline temp

Takeaway

Class encapsulates functionality and data

Moving features between objects

- Move methods and fields
- Extract class

Takeaway

Organizing data

- Properly organized and associated classes are more portable and reusable

Multiple techniques

- Encapsulate fields and collections
- Replace strings for Enums
- Replace magic numbers with symbolic constants
- Change type field with class

Takeaway

Simplify conditional expressions

- Consolidate conditional expression
- Consolidate duplicate conditional fragments
- Decompose conditional

Takeaway

Simplify method calls

- Preserve whole object
- Introduce parameter object
- Split and merge methods

Takeaway

Dealing with generalization

- Moving functionality along the class inheritance hierarchy
- Pull up or down fields and methods
- Extract interface, subclass, or super class

Takeaway

Rename

- Use the IDE functionality
- Manually, but with care