Refactoring Fundamentals

Class and Object Refactorings

Steve Smith
Ardalis.com
@ardalis

pluralsight
hardcore developer training

In This Course

- ~~What is Refactoring?~~
- ~~Why do it?~~
- ~~What's the process?~~
- ~~What are some tools that can assist with it?~~
- ~~What is a *Code Smell*?~~
- ~~*What are some examples of Code Smells?*~~
- What are some common refactorings?
- How does one apply them correctly?

Class-Related Refactorings

- Encapsulate Field
- Encapsulate Collection
- Move Field
- Move Method
- Extract Class
- Inline Class
- Extract Interface
- Extract Subclass
- Extract Superclass
- Hide Delegate
- Remove Middle Man

SOLID Principles of Object Oriented Design

This course introduces foundational principles of creating well-crafted code and is appropriate for anyone hoping to improve as a developer

Encapsulate Field

There is a public field.

- **Create get and set methods for the field**
  - Alternately, replace field with automatic properties
- **Find all clients outside the class that reference the field**
- **Replace each reference with a reference to the new property accessor**

- ***Compile and Test after each change***
- **Change the field to be *private***
- ***Compile and Test***

Encapsulate Field

```
public class Customer
{
    public string Name;
}

public class Customer
{
    private string _name;
    public string Name {
        get { return _name; }
        set { _name = value; }
    }
}

public class Customer
{
    public string Name { get; set; }
}
```

Encapsulate Collection

A method (especially a property) returns a collection.

- **Add methods for adding and removing items to/from collection**
- **Initialize the field to an empty collection**
- ***Compile***
- **Find references that set the collection**
  - Modify to use the add and remove methods instead
- ***Compile and Test* after each change**
- **Find references that get the collection and then modify its contents**
  - Modify to use the add and remove methods instead
- ***Compile and Test* after each change**
- **Modify the getter to return a read-only collection**
- ***Compile and Test***

Encapsulate Collection

```
public class Student
{
    public string Name { get; set;}
    public List<Class> Classes;
}

public class RegistrationService
{
    public void RegisterStudent(Student student, Class theClass)
    {
        var currentClasses = student.Classes;
        // perform some logic to confirm the student can enroll in this class
        currentClasses.Add(theClass);
    }
}
```

Encapsulate Collection

```
public class Student
{
    private List<Class> _classes = new List<Class>();

    public string Name { get; set;}
    public IReadOnlyCollection<Class> Classes
    {
        get
        {
            return _classes.AsReadOnly();
        }
    }

    public void EnrollIn(Class theClass)
    {
       // perform some logic to confirm the student can enroll in this class
       _classes.Add(theClass);
    }
}
```

Encapsulate Collection

```
public class RegistrationService
{
    public void RegisterStudent(Student student, Class theClass)
    {
        student.EnrollIn(theClass);
    }
}
```

Move Field

A field is, or will be, used by another class more than the class on which it is defined.

- If the field is public, first Encapsulate Field
- Create a field in the target class with get/set methods
- Compile
- Determine how to reference the target object from the source
- Remove the field from the source class
- Replace references to the source field with references to the appropriate method on the target
- *Compile and Test*

Move Field

```
public class Account
{
    private AccountType _type;
    private decimal _interestRate;

    // other methods that use _interestRate

    public decimal CalculateInterest(decimal amount, int days)
    {
        return amount * _interestRate * days / 365;
    }
}
```

Move Field

```
public class AccountType
{
    public decimal InterestRate { get; private set; }
}

public class Account
{
    private AccountType _type;

    // other methods that use InterestRate

    public decimal CalculateInterest(decimal amount, int days)
    {
        return amount * _type.InterestRate * days / 365;
    }
}
```

Move Method

A method is, or will be, using or used by more features of another class than the class on which it is defined.

- **Examine all fields/methods used by method defined in source class**
  - Consider moving them all
- **Check sub- and superclasses for the method**
- **Declare the method on the target class**
  - Copy the code from the source method and adjust to make it work
- ***Compile***
- **Determine how to reference the target object from the source**
- **Delegate from the old method to the new one**
- **Compile and Test**

Moving methods is the bread and butter of refactoring.

Martin Fowler

Move Method

```
public class Cart
{
    private readonly Customer _customer;

    public void Checkout()
    {
        // do other things
        string orderId = "";
        EmailCheckoutNotification(_customer.EmailAddress, orderId);
    }

    private void EmailCheckoutNotification(string emailAddress, string
    orderId)
    {
        // send an email with the order id to the customer
    }
}
```

Move Method

```
public class NotificationService
{
    public void EmailCheckoutNotification(string emailAddress, string
    message)
    {
        // send an email with the order id to the customer
    }
}
```

Move Method

```
public class Cart
{
    private readonly Customer _customer;
    private NotificationService _notificationService = new
    NotificationService();

    public void Checkout()
    {
        // do other things
        string orderId = "";
        EmailCheckoutNotification(_customer.EmailAddress, orderId);
    }

    private void EmailCheckoutNotification(string emailAddress, string
    orderId)
    {
        _notificationService.EmailCheckoutNotification(emailAddress,
    orderId);
    }
}
```

Move Method

```
public class Cart
{
    private readonly Customer _customer;
    private NotificationService _notificationService = new
    NotificationService();

    public void Checkout()
    {
        // do other things
        string orderId = "";

    _notificationService.EmailCheckoutNotification(_customer.EmailAddress,
    orderId);
    }
}
```

Move Method

```
public class Cart
{
    private readonly Customer _customer;
    private NotificationService _notificationService;

    public Cart(NotificationService notificationService)
    {
       _notificationService = notificationService;
    }

    public Cart() : this(new NotificationService()) {}
    public void Checkout()
    {
        // do other things
        string orderId = "";

    _notificationService.EmailCheckoutNotification(_customer.EmailAddress,
    orderId);
    }
}
```

Move Method

```
public class Cart
{
    private readonly Customer _customer;
    private INotificationService _notificationService;

    public Cart(INotificationService notificationService)
    {
       _notificationService = notificationService;
    }

    public Cart() : this(new NotificationService()) {}
    public void Checkout()
    {
        // do other things
        string orderId = "";

    _notificationService.EmailCheckoutNotification(_customer.EmailAddress,
    orderId);
    }
}
```

Extract Class

You have one class doing work that should be done by two.

- **Determine how to split current class's responsibilities**
- **Create a new class to hold split-off responsibility**
- **Make a link from the old class to the new one**
  - Prefer a one-way link if possible
- **Use Move Field on each field you wish to move**
- **Use Move Method on each method you wish to move**
- ***Compile and Test* after each move**
- **Decide whether to expose the new class**

Extract Class

```
public class Person
{
    public string FirstName { get; set; }
    public string MiddleName { get; set; }
    public string LastName { get; set; }
    public DateTime DateOfBirth { get; set; }
    // other properties

    public string FullName()
    {
        return (FirstName + " " + MiddleName).Trim() + " " + LastName;
    }
}
```

Extract Class

```
public class Name
{
    public Name(string firstName, string middleName, string lastName)
    {
        this.FirstName = firstName;
        this.MiddleName = middleName;
        this.LastName = lastName;
    }

    public string FirstName { get; private set; }
    public string MiddleName { get; private set; }
    public string LastName { get; private set; }

    public string FullName()
    {
        return (FirstName + " " + MiddleName).Trim() + " " + LastName;
    }
}
```

Extract Class

```
public class Person
{
    public Name Name { get; set; }
    public DateTime DateOfBirth { get; set; }
    // other properties
}
```

Inline Class

A class isn't *doing* very much.

- **Declare the public interface of the source class on the absorbing class**
  - Delegate from the absorbing class to the source class for now
- **Change all references to the source class to the absorbing class**
- ***Compile and Test***
- **Use Move Method and Move Field to move features from the source to the absorbing class until there is nothing left**
- **Hold a short, simple funeral service**
- **Delete the source class**

Inline Class

```
public class Name
{
    public Name(string firstName, string middleName, string lastName)
    {
        this.FirstName = firstName;
        this.MiddleName = middleName;
        this.LastName = lastName;
    }

    public string FirstName { get; private set; }
    public string MiddleName { get; private set; }
    public string LastName { get; private set; }

    public string FullName()
    {
        return (FirstName + " " + MiddleName).Trim() + " " + LastName;
    }
}
```

Inline Class

```
public class Person
{
    public string FirstName { get; set; }
    public string MiddleName { get; set; }
    public string LastName { get; set; }
    public DateTime DateOfBirth { get; set; }

    public string FullName()
    {
        return (FirstName + " " + MiddleName).Trim() + " " + LastName;
    }
}
```

Extract Interface

Several clients use the same subset of a class's interface, or two classes have part of their interfaces in common.

- Create an empty interface
- Declare the common operations in the interface
- Declare the class(es) as implementing the interface
- Adjust clients to depend on the interface
- *Compile and Test*

Extract Interface

```
public class NotificationService
{
    public void EmailCheckoutNotification(string emailAddress, string message)
    {
        // send an email with the order id to the customer
    }
}

public class Cart
{
    private readonly Customer _customer;
    private NotificationService _notificationService = new NotificationService();

    public void Checkout()
    {
        // do other things
        string orderId = "";
        _notificationService.EmailCheckoutNotification(_customer.EmailAddress, orderId);
    }
}
```

Extract Interface

```
public interface INotificationService
{
    void EmailCheckoutNotification(string emailAddress, string message);
}
public class NotificationService : INotificationService
{
    public void EmailCheckoutNotification(string emailAddress, string message)
    {
        // send an email with the order id to the customer
    }
}
public class Cart
{
    private readonly Customer _customer;
    private INotificationService _notificationService = new NotificationService();

    public void Checkout()
    {
        // do other things
        string orderId = "";
        _notificationService.EmailCheckoutNotification(_customer.EmailAddress, orderId);
    }
}
```

Extract Subclass

A class has features that are used only in some instances.

- **Define a new subclass of the source class**
- **Provide constructors for the new subclass**
  - Alternately use Replace Constructor with Factory Method
- **Find all calls to constructors of the superclass and replace with calls to the new constructor if appropriate**
- **Use Push Down Method and Push Down Field to move features to the subclass**
- **Replace type codes with constants**
  - Refactor clients of these fields with Replace Conditional with Polymorphism
- ***Compile and Test* after each change**

Extract Subclass

```
public class AdCreative
{
    public bool IsImage { get; set; }
    public string AdText { get; set; }
    public int Height { get; set; }
    public int Width { get; set; }
    public DateTime DateCreated { get; set; }

    public string Render()
    {
        if (IsImage)
        {
            return String.Format("<img height='{0}' width='{1}' alt='{2}'
   />",
                Height, Width, AdText);
        }
        return AdText;
    }
}
```

Extract Subclass

```
public class ImageAdCreative : AdCreative
{
    public bool IsImage { get { return true; } }
    public int Height { get; set; }
    public int Width { get; set; }

    public override string Render()
    {
        return String.Format("<img height='{0}' width='{1}' alt='{2}' />",
            Height, Width, AdText);
    }
}
```

Extract Subclass

```
public class AdCreative
{
    public bool IsImage { get; set; }
    public string AdText { get; set; }
    public DateTime DateCreated { get; set; }

    public virtual string Render()
    {
        return AdText;
    }
}
```

Extract Superclass

You have two classes with similar features.

- **Create an empty abstract superclass; make the original classes inherit from this new class**
- **Use Pull Up Field, Pull Up Method, and Pull Up Constructor Body to move common elements to the superclass**
- ***Compile and Test* after each operation**
- **Check all clients of the original classes**
  - If they only use the common behavior, have them reference the new superclass
- ***Compile and Test***

Hide Delegate

A client is calling a delegate class of an object.

- For each method on the delegate class, create a simple delegating method
- Adjust the client to call the new delegating method
- *Compile and Test* after each adjustment
- If no client needs to access the delegate object, remove its accessor
- *Compile and Test*

Hide Delegate

```
public class Department
{
    public string Name { get; set; }
    public Employee Manager { get; set; }
}

public class Employee
{
    public string Name { get; set; }
    public Department Department { get; set; }
}

public class Client
{
    public void DoSomething(Employee employee)
    {
        var manager = employee.Department.Manager;
    }
}
```

Hide Delegate

```
public class Employee
{
    public string Name { get; set; }
    public Department Department { get; set; }
    public Employee Manager
    {
        get
        {
            return this.Department.Manager;
        }
    }
}

public class Client
{
    public void DoSomething(Employee employee)
    {
        var manager = employee.Manager;
    }
}
```

Remove Middle Man

A class is doing too much delegation.

- **Create an accessor for the delegate class (or make it public)**
- **For each client using a delegate method**
  - Remove the method from the server
  - Update the client to call the delegate class directly
- ***Compile and Test* after each update**

Remove Middle Man

```
public class Employee
{
    public string Name { get; set; }
    public Department Department { get; set; }
    public Employee Manager
    {
        get
        {
            return this.Department.Manager;
        }
    }
}

public class Client
{
    public void DoSomething(Employee employee)
    {
        var manager = employee.Manager;
    }
}
```

Remove Middle Man

```
public class Department
{
    public string Name { get; set; }
    public Employee Manager { get; set; }
}

public class Employee
{
    public string Name { get; set; }
    public Department Department { get; set; }
}

public class Client
{
    public void DoSomething(Employee employee)
    {
        var manager = employee.Department.Manager;
    }
}
```

Summary

- Encapsulate Field
- Encapsulate Collection
- Move Field
- Move Method
- Extract Class
- Inline Class
- Extract Interface
- Extract Subclass
- Extract Superclass
- Hide Delegate
- Remove Middle Man

References

Books

Refactoring http://amzn.to/110tscA

Web

Refactoring Catalog http://www.refactoring.com/catalog/

Thanks!

Steve Smith
Ardalis.com
Twitter: @ardalis
To Teach Is To Learn Twice