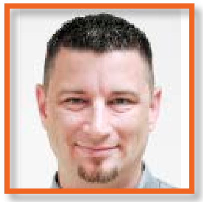
Refactoring Fundamentals

Class Hierarchy 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

- Pull Up Field
- Push Down Field
- Pull Up Method
- Push Down Method
- Collapse Hierarchy
- Replace Inheritance with Delegation
- Replace Delegation with Inheritance
- Replace Type Code with Class
- Replace Type Code with Subclasses
- Replace Conditional with Polymorphism

Pull Up Field

Two subclasses have the same field.

- **Inspect all uses of the candidate fields**
  - Ensure they are being used in the same way
- **Apply Rename Field to one field, if necessary**
- ***Compile and Test***
- **Create a new field in the superclass**
  - If the fields were *private*, this one will need to be *protected*
- **Delete the fields from the subclasses**
- ***Compile and Test***

Pull Up Field

```
public class Grid : CustomWebControl
{
    private HttpContext _context;
}

public class DatePicker : CustomWebControl
{
   private HttpContext _httpContext;
}
```

Pull Up Field

```
public class Grid : CustomWebControl
{
    private HttpContext _context;
}

public class DatePicker : CustomWebControl
{
   private HttpContext _context;
}

public class CustomWebControl
{
   protected HttpContext _context;
}
```

Pull Up Field

```
public class Grid : CustomWebControl
{
}

public class DatePicker : CustomWebControl
{
}

public class CustomWebControl
{
    protected HttpContext _context;
}
```

Push Down Field

A field is used only by some subclasses.

- **Declare the field in all subclasses**
  - Mark it *private* rather than *protected* if applicable
- **Remove the field from the superclass**
- ***Compile and Test***
- **Remove the field from all subclasses that do not need it**
- ***Compile and Test***

Push Down Field

```
public class Grid : CustomWebControl
{
    public void Render()
    {
       for(int i=0;i<Rows;i++)
       {
          for(int j=0;i<Columns;i++) {}
        }
    }
}

public class RichTextBox: CustomWebControl
{
}

public class CustomWebControl
{
    protected HttpContext Context;
    protected int Columns;
}
```

Push Down Field

```
public class Grid : CustomWebControl
{
    private int Columns;
    public void Render()
    {
       for(int i=0;i<Rows;i++)
       {
          for(int j=0;i<Columns;i++) {}
        }
    }
}
public class RichTextBox: CustomWebControl
{
    private int Columns;
}

public class CustomWebControl
{
    protected HttpContext Context;
}
```

Push Down Field

```
public class Grid : CustomWebControl
{
    private int Columns;
    public void Render()
    {
       for(int i=0;i<Rows;i++)
       {
          for(int j=0;i<Columns;i++) {}
       }
    }
}
public class RichTextBox: CustomWebControl
{
}

public class CustomWebControl
{
    protected HttpContext Context;
}
```

Pull Up Method

You have methods with identical results on subclasses.

- **Inspect the methods to ensure they are identical**
  - You can use Substitute Algorithm to force them to be identical
- **Adjust the signatures to be consistent**
- **Create a new superclass method**
  - Copy the body of one of the subclass methods into this method
  - If the method calls methods only available on subclasses, declare an abstract or virtual method in the superclass
- **Delete one subclass method**
- ***Compile and Test***
- **Repeat until only the superclass method remains**
- **Consider modifying callers to require the superclass type**

Pull Up Method

```
public class Grid : CustomWebControl
{
   public void Render() {
      // calls ApplyStyles()
   }
   private string ApplyStyles(object element) {}
}

public class RichTextBox: CustomWebControl
{
   public void Render() {
      // calls ApplyStyles()
   }
   private string ApplyStyles(object element) {}
}

public class CustomWebControl
{
}
```

Pull Up Method

```
public class Grid : CustomWebControl
{
    protected override string ApplyStyles(object element) {}
}

public class RichTextBox: CustomWebControl
{
    protected override string ApplyStyles(object element) {}
}

public abstract class CustomWebControl
{
    public void Render() {
       // calls ApplyStyles()
    }
    protected abstract string ApplyStyles(object element);
}
```

Push Down Method

Behavior on a superclass is relevant only for some of its subclasses.

- **Declare the method in all subclasses**
  - Copy the body of the superclass method into each subclass method
  - Adjust private fields in the superclass that are used by the method
- **Delete the method from the superclass**
  - Callers may need to be updated to use the subclass type
- ***Compile and Test***
- **Remove the method from each subclass that does not need it**
- ***Compile and Test* after each removal**

Push Down Method

```
// Grid, RichTextBox, etc.
public class VisibleControl: CustomWebControl
{
    public void Render()
    {
       ApplyStyles("");
    }
}

// hidden form field, other behind-the-scenes controls
public class InvisibleControl: CustomWebControl
{
    public void Render()
    {
    }
}

public abstract class CustomWebControl
{
    protected string ApplyStyles(object element) {}
}
```

Push Down Method

```
// Grid, RichTextBox, etc.
public class VisibleControl: CustomWebControl
{
    public void Render()
    {
       ApplyStyles("");
    }
    protected string ApplyStyles(object element) {}
}

// hidden form field, other behind-the-scenes controls
public class InvisibleControl: CustomWebControl
{
    public void Render()
    {
    }
    protected string ApplyStyles(object element) {}
}

public abstract class CustomWebControl
{
    protected string ApplyStyles(object element) {}
}
```

Push Down Method

```
// Grid, RichTextBox, etc.
public class VisibleControl: CustomWebControl
{
    public void Render()
    {
       ApplyStyles("");
    }
    protected string ApplyStyles(object element) {}
}

// hidden form field, other behind-the-scenes controls
public class InvisibleControl: CustomWebControl
{
    public void Render()
    {
    }
    protected string ApplyStyles(object element) {}
}

public abstract class CustomWebControl
{
}
```

Push Down Method

```
// Grid, RichTextBox, etc.
public class VisibleControl: CustomWebControl
{
    public void Render()
    {
       ApplyStyles("");
    }
    protected string ApplyStyles(object element) {}
}

// hidden form field, other behind-the-scenes controls
public class InvisibleControl: CustomWebControl
{
    public void Render()
    {
    }
}

public abstract class CustomWebControl
{
}
```

Collapse Hierarchy

A superclass and subclass are not very different.

- **Choose which class to remove (subclass or superclass)**
- **Move behavior to the class that will remain**
  - Pull Up Field, Pull Up Method to the superclass, or
  - Push Down Field, Push Down Method to the subclass
- ***Compile and Test* as part of each move**
- **Adjust references to the class that will be removed to use the merged class**
  - Variable declarations
  - Parameters
  - Constructors
- **Remove the empty class**
- ***Compile and Test***

Collapse Hierarchy

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

public class Manager : Employee
{
    // all custom behavior has been moved elsewhere
}
```

Collapse Hierarchy

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

public class Manager : Employee
{
    public IReadOnlyList<Employee> DirectReports { get; set; }

    public decimal CalculateBonus()
    {
        return 0.10 * Salary;
    }
}
```

Collapse Hierarchy

```
public class Employee
{
    public string Name { get; set; }
    public decimal Salary { get; set; }
    public Employee Manager { get; set; }
    public IReadOnlyList<Employee> DirectReports { get; set; }

    public decimal CalculateBonus()
    {
        return 0.10m * Salary;
    }
}

public class Manager : Employee
{
}
```

Collapse Hierarchy

```
public class Employee
{
    public string Name { get; set; }
    public decimal Salary { get; set; }
    public Employee Manager { get; set; }
    public IReadOnlyList<Employee> DirectReports { get; set; }

    public decimal CalculateBonus()
    {
       if(DirectReports.Any())
        {  return 0.10m * Salary;  }
      return 0m;
    }
}
```

Collapse Hierarchy

```
public class Employee
{
    public string Name { get; set; }
    public decimal Salary { get; set; }
    public Employee Manager { get; set; }
    public IReadOnlyList<Employee> DirectReports { get; set; }
   public bool IsManager { get; set; }

    public decimal CalculateBonus()
    {
        if(IsManager)
        {  return 0.10m * Salary;  }
       return 0m;
    }
}
```

Collapse Hierarchy

```
public class Employee
{
    public string Name { get; set; }
    public decimal Salary { get; set; }
    public Employee Manager { get; set; }
    public IReadOnlyList<Employee> DirectReports { get; set; }
}

public class BonusCalculator
{
    public decimal CalculateBonusFor(Employee employee)
    {
        // perform calculations here
    }
}
```

Replace Inheritance with Delegation

A subclass uses only part of a superclass's interface
or does not want to inherit data.

- **Create a field in the subclass with the type of the superclass**
  - Initialize the field to a new instance
- **Change each method defined in the subclass to use this new field**
- ***Compile and Test* after each change**
  - Note that you won't be able to do this for methods that themselves invoke a method on the base class, yet
- **Remove the subclass declaration**
- **For each superclass method used by a client, add a simple delegating method**
- ***Compile and Test***

Replace Inheritance with Delegation

```
public class Company : List<Employee>
{
    public IEnumerable<Employee> ListCurrentEmployees()
    {
        return this.Where(e => e.IsEmployed).AsEnumerable();
    }
    public void Hire(Employee employee)
    {
        this.Add(employee);
    }
    public void Fire(Employee employee)
    {
        this.Remove(employee);
    }
}
```

Replace Inheritance with Delegation

```
public class ClientCode
{
    public void Main()
    {
        var company = new Company();

        var employees = company.ListCurrentEmployees();

        var employee = new Employee();

        // ok
        company.Hire(employee);

        // ok
        company.Fire(employee);

        // should not be allowed
        company.Clear();
    }
}
```

Replace Inheritance with Delegation

```
public class Company : List<Employee>
{
    private List<Employee> _employees = new List<Employee>();

    public IEnumerable<Employee> ListCurrentEmployees()
    {
        return this.Where(e => e.IsEmployed).AsEnumerable();
    }
    public void Hire(Employee employee)
    {
        this.Add(employee);
    }
    public void Fire(Employee employee)
    {
        this.Remove(employee);
    }
}
```

Replace Inheritance with Delegation

```
public class Company : List<Employee>
{
    private List<Employee> _employees = new List<Employee>();

    public IEnumerable<Employee> ListCurrentEmployees()
    {
        return _employees.Where(e => e.IsEmployed).AsEnumerable();
    }
    public void Hire(Employee employee)
    {
        _employees.Add(employee);
    }
    public void Fire(Employee employee)
    {
        _employees.Remove(employee);
    }
}
```

Replace Inheritance with Delegation

```
public class Company
{
    private List<Employee> _employees = new List<Employee>();

    public int Count { get { return _employees.Count; } }

    public IEnumerable<Employee> ListCurrentEmployees()
    {
        return _employees.Where(e => e.IsEmployed).AsEnumerable();
    }
    public void Hire(Employee employee)
    {
        _employees.Add(employee);
    }
    public void Fire(Employee employee)
    {
        _employees.Remove(employee);
    }
}
```

Replace Delegation with Inheritance

You're using delegation and are often writing many simple delegations for the entire interface.

- **Make the delegating object a subclass of the delegate object**
- ***Compile***
- **Remove all simple delegation methods**
- ***Compile and Test***
- **Replace all other delegations with calls to the object itself**
  - That is, replace references to the field with references to "this"
- **Remove the delegate field**
- ***Compile and Test***

Replace Delegation with Inheritance

```
public class EmployeeCollection
{
    private List<Employee> _employees = new List<Employee>();

    public int Count { get { return _employees.Count; } }

    public IEnumerable<Employee> ListCurrentEmployees()
    {
        return _employees.Where(e => e.IsEmployed).AsEnumerable();
    }
    public void Add(Employee employee)
    {
        _employees.Add(employee);
    }
    public void Remove(Employee employee)
    {
        _employees.Remove(employee);
    }
}
```

Replace Delegation with Inheritance

```
public class EmployeeCollection : List<Employee>
{
    private List<Employee> _employees = new List<Employee>();

    public int Count { get { return _employees.Count; } }

    public IEnumerable<Employee> ListCurrentEmployees()
    {
        return _employees.Where(e => e.IsEmployed).AsEnumerable();
    }
    public void Add(Employee employee)
    {
        _employees.Add(employee);
    }
    public void Remove(Employee employee)
    {
        _employees.Remove(employee);
    }
}
```

Replace Delegation with Inheritance

```
public class EmployeeCollection : List<Employee>
{
    private List<Employee> _employees = new List<Employee>();

    public IEnumerable<Employee> ListCurrentEmployees()
    {
        return _employees.Where(e => e.IsEmployed).AsEnumerable();
    }
}
```

Replace Delegation with Inheritance

```
public class EmployeeCollection : List<Employee>
{
    private List<Employee> _employees = new List<Employee>();

    public IEnumerable<Employee> ListCurrentEmployees()
    {
        return this.Where(e => e.IsEmployed).AsEnumerable();
    }
}
```

Replace Delegation with Inheritance

```
public class EmployeeCollection : List<Employee>
{
    public IEnumerable<Employee> ListCurrentEmployees()
     {
         return this.Where(e => e.IsEmployed).AsEnumerable();
     }
}
```

Replace Type Code with Class

A class has a numeric type code that does not affect its behavior.

- **Create a new class for the type code**
  - Give it a numeric field that matches the type code
  - Give it several static variables of its type that represent each valid option
  - Give it a static method for returning an instance given a numeric type code
- **Modify the original class to use this new class**
- ***Compile and Test***
- **One by one, copy and modify the original class methods to use the new class**
  - Methods that took a type code as an argument now take the new class as an argument
  - Methods that returned a type code now return the new class
  - Update clients to use these new methods
- ***Compile and Test*** **after each update**
- **Remove the old methods and the original type codes**
- ***Compile and Test***

Replace Type Code with Subclasses

You have an immutable type code that affects the behavior of a class.

- **Replace Constructor with a Factory Method**
  - i.e. void Customer(int type) becomes Customer Create(int type)
- **For Each Value of Type Code, Create a Subclass**
  - Hardcode the get code property to return the expected value
- ***Compile and Test*** **after each type code replacement**
- **Remove the type code field from the superclass**
- **Declare accessors for the type code as abstract**
- ***Compile and Test***

Replace Conditional with Polymorphism

You have a conditional that chooses different behavior depending on the type of an object.

- **If necessary, use Extract Method to get the conditional in its own method**
- **If necessary, use Move Method to pull the conditional to the top of the inheritance hierarchy**
- **Choose one subclass, override the conditional method**
  - Copy the body of the conditional into the subclass's method
- ***Compile and Test***
- **Remove that leg of the conditional**
- ***Compile and Test***
- **Repeat with each leg until all have been turned into subclass methods**
- **Make the superclass method abstract**

Summary

- Pull Up Field
- Push Down Field
- Pull Up Method
- Push Down Method
- Collapse Hierarchy
- Replace Inheritance with Delegation
- Replace Delegation with Inheritance
- Replace Type Code with Class
- Replace Type Code with Subclasses
- Replace Conditional with Polymorphism

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

pluralsight
hardcore developer training