Refactoring Fundamentals

Pattern-Based 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?

Pattern-Based Refactorings

- Encapsulate Classes with Factory
- Form Template Method
- Introduce Null Object
- Move Accumulation to Visitor
- Move Embellishment to Decorator
- Replace Conditional Dispatcher with Command
- Replace Conditional Logic with Strategy
- Replace State-Altering Conditionals with State
- Replace Type Code with State/Strategy
- Unify Interfaces with Adapter

Design Patterns Library

A reference library for design patterns of all types

Encapsulate Classes with Factory

Clients directly instantiate classes that reside in one package and implement a common interface.

- **Identify classes to encapsulate that share a common interface**
- **Find a client that instantiates one of these classes via its constructor**
  - Use Extract Method on the constructor call to create a static Factory method
  - Now use Move Method to add this method to the superclass
- ***Compile and Test***
- **Update all other callers of the constructor to use the new Factory method**
- ***Compile and Test***
- **Repeat for any other constructors the class exposes**
- **Declare the class's constructor(s) to be non-public**
- ***Compile***
- **Repeat the above steps for all classes you wish to encapsulate**

Form Template Method

Two methods in subclasses perform similar steps in the same order, yet the steps are different.

- Decompose the methods using Extract Method
  - Each method pulled out should be identical, or completely different
- Use Pull Up Method to pull the identical methods into the superclass
- Use Rename Method to make the method signatures at each step match
- *Compile and Test* after each signature change
- Use Pull Up Method on one of the original methods
- Define the signatures of the different methods as abstract on the superclass
- *Compile and Test*
- Remove the other methods
- *Compile and Test* after each removal

Introduce Null Object

You have repeated checks for a null value.

- **Create a subclass of the source class to act as a null version of the class**
  - Create an IsNull property on the source class
  - Always return false from the base class, and true from the new null class
- ***Compile***
- **Find all places that can return null when asked for the source object**
  - Return the new null object instead
- **Find all places that check the returned type for null, and replace them with calls to IsNull**
- ***Compile and Test***
- **Look for cases where a client takes one action if null, and another if not**
  - Override the operation in the null class with the behavior that should occur when the class is null
- **Remove the condition checks for clients that use the overridden behavior**
- ***Compile and Test***

http://en.wikipedia.org/wiki/Tony_Hoare

Move Accumulation to Visitor

A method accumulates information from heterogeneous classes.

- **Promote reused local variables in the method to host fields**
- ***Compile and Test***
- **Apply Extract Method on the accumulation logic for one source class**
  - Have the method accept an argument of the source class's type
  - Name the method Accept*SourceClassName*
- **Repeat this step for each source class**
- **Apply Extract Method on the body of an Accept… to produce a new method, Visit*SourceClassName***
  - The new method will take one argument of the source class's type
- **Apply Move Method to move each Accept… method to its source class**
- ***Compile and Test***

Move Accumulation to Visitor (cont.)

- **In the accumulation method, apply Inline Method on every call to an Accept… method**
- ***Compile and Test***
- **Adjust the interfaces of the accumulation sources so the Accept… methods may be called polymorphically**
- **Generalize the accumulation method to call the Accept… methods polymorphically for every accumulation source**
- ***Compile and Test***
- **Apply Extract Interface on the host to produce a *visitor interface***
- **Change the signature on every Accept method so it uses the visitor interface**
- ***Compile and Test***

Move Embellishment to Decorator

Code provides an embellishment to a class's core responsibility.

- **Identify or create an *enclosure type* – a type that declares the public interface of the class being embellished**
- **Locate the conditional that adds embellishment and remove it using Replace Conditional with Polymorphism**
  - If you create a factory method here, be sure its return type is the enclosure type
  - If you have logic that must occur before/after the embellishment code, use Form Template Method
- ***Compile and Test***
- **In the above step, one or more subclasses were created. Transform these into delegating classes using Replace Inheritance with Delegation**
  - Make each delegating class implement the enclosure type
  - Make the type of the delegating class's delegate field be the enclosure type
  - Decide whether the embellishment code will be called before or after the delegate code
  - Make sure any method that returns the embellished class decorates it first

Move Embellishment to Decorator (cont.)

- *Compile and Test*
- **Each delegating class assigns its delegate field to a new instance of the embellished class in its constructor**
    - Apply Extract Parameter to the instantiation of the embellished class, creating a constructor parameter
- *Compile and Test*

Replace Conditional Dispatcher with Command

Conditional logic is used to dispatch requests and execute actions.

- **Locate the conditional that is dispatching work**
  - Apply <u>Extract Method</u> on each leg of the conditional
  - Repeat until all legs are simply calling methods
- ***Compile and Test***
- **Apply <u>Extract Class</u> on each of the above methods to produce a concrete Command class**
  - Add the method to the new class and make it public
- ***Compile and Test***
- **Define a Command interface or abstract class and modify each concrete Command class above to implement it**
  - Ideally, it should simply have one public void Execute() method
- **Modify all client code to work with the Command type**
- **On the class with the conditional dispatcher, define a *command map***
  - A collection of instances, keyed by a unique identifier (e.g. command name)
- **Replace the conditional code with code to fetch the correct command and execute it**
- ***Compile and Test***

Replace Conditional Logic with Strategy

Conditional logic in a method controls which of several variants of a calculation are executed.

- Identify the *context* class
- Create a new concrete Strategy class named for the behavior the conditional logic performs
- Use Move Method to move the conditional method to the new Strategy class
  - Move any helper methods as well
- *Compile and Test*
- Let clients pass an instance of the Strategy to the context using Extract Parameter
- *Compile and Test*
- Apply Replace Conditional with Polymorphism on the Strategy's calculation method
  - This should yield several subclasses, one for each leg of the conditional
  - If possible, make the original Strategy an abstract class
- *Compile and Test*

Replace State-Altering Conditionals with State

The conditional expressions that control an object's state transitions are complex.

- **Identify the object (called the context here) that has the state field**
- **Apply Replace Type Code with Class so the state field becomes a class**
  - We'll refer to this as the State base class
  - Each constant in the context refers to an instance of this base class
- ***Compile***
- **Make each context state constant an instance of a specific subclass of the State base class**
- ***Compile***
- **Find context methods that change the value of the original state field**
  - Copy these methods to the State base class (pass in the context class if necessary to get them to work)
- ***Compile and Test***
- **Choose a state the context can enter, and copy any methods that make this state transition from the State base class to the appropriate subclass (override the base)**
  - Remove any unnecessary logic, such as verifications of current state
  - Repeat for each state
  - Delete the bodies of these methods from the base class
- ***Compile and Test***

Replace Type Code with State (or Strategy)

You have a type code that affects the behavior of a class,
but you cannot use subclassing.

- **Encapsulate the type code within its class**
- **Create a new class to represent the State**
  - Name it after the type code's intent
- **Create one subclass for each option the type code can have**
- **Create an abstract query method in the state object to return the type code**
  - Hard code each subtype to return the type code representing its state
- ***Compile***
- **Now create a field in the original class to hold the state object**
- **Adjust the type query in the original class to delegate to the state object**
- **Adjust the type setting methods to assign an instance of the appropriate state subclass to the state object**
- ***Compile and Test***

Unify Interfaces with Adapter

Clients interact with two classes,
one of which has a preferred interface.

- **Identify which of the two classes' interfaces the client prefers to use**
  - □ Extract this interface into a common interface
  - □ Update any of this class's methods that have arguments of its type to use the new interface type instead
- ***Compile and Test***
- **Extract Class on the client code calling the non-preferred class's code**
  - □ Create a simple class with a field to hold the non-preferred class and a way to populate this field, as well as a property getter for it
  - □ This is the *Adapter* class
- **Now update all of the client's fields, local variables, parameters, etc. from the non-preferred class type to the adapter type**
  - □ Adjust them to reference the class via the its getter
- ***Compile and Test***

Unify Interfaces with Adapter (cont.)

Clients interact with two classes,
one of which has a *preferred* interface.

- **Perform <u>Extract Method</u> on each client invocation**
  - Add the method to the adapter
  - Parameterize the method with an instance of the non-preferred class if necessary
- ***Compile and Test. Repeat for all non-preferred class methods.***
- **Now use <u>Move Method</u> to move methods to the adapter**
  - Take care to try and keep the method signatures matching the common interface
- ***Compile and Test***
- **Update the adapter to actually implement the common interface.**
  - Change any adapter methods that had arguments of its class type to use the common interface type
- ***Compile and Test***
- **Update the client class so all types that used the adapter now use the common interface**
- ***Compile and Test***

Summary

- Encapsulate Classes with Factory
- Form Template Method
- Introduce Null Object
- Move Accumulation to Visitor
- Move Embellishment to Decorator
- Replace Conditional Dispatcher with Command
- Replace Conditional Logic with Strategy
- Replace State-Altering Conditionals with State
- Replace Type Code with State/Strategy
- Unify Interfaces with Adapter

References

Books

Refactoring http://amzn.to/110tscA

Refactoring to Patterns http://amzn.to/Vq5Rj2

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