Refactoring (continued)

Source:
"Refactoring: Improving the Design of Existing Code", Martin Fowler

Effective Refactoring

- Knowing what refactorings are available
- Knowing when to apply them

Refactoring Catalog

- Example: Introduce Parameter Object

Knowing When to Refactor

"If it stinks, change it."

Grandma Beck, discussing child-rearing philosophy

Bad Smells in Code (Signs that you need to refactor)

- Duplicated code
- Long method
- Large class
- Long parameter list
- Divergent change
- Shotgun surgery
- Feature envy
- Data clumps
- Primitive obsession
- Switch statements
- Parallel inheritance hierarchies
- Lazy class
- Speculative generality
- Temporary field
- Message chains
- Middle man
- Inappropriate intimacy
- Alternative classes with different interfaces
- Incomplete library class
- Data class
- Refused bequest
- Comments

Duplicated Code

- The same code structure is duplicated in multiple places
  - Identical sections of code
  - Similar sections of code (e.g., methods with similar structures)
- Hard to maintain, serious design problem

- Same Class => Extract Method
- Sibling Classes => Extract Method, Pull Up Method
- Similar Code In Sibling Classes => Form Template Method
- Unrelated Classes => Extract Class, all classes invoke the new class
- Unrelated Classes => Extract Method in one class, make other classes call its method

Temporary Field

- A class has one or more fields (i.e., variables) that are not used all the time
- Trying to understand why and when these fields aren't set is confusing

- Example: Some instances have a particular attribute, some don't
  - E.g., Employee class with hourlyRate field that is used only for some employees
    - Missing subclass. Use <u>Extract Subclass</u> to push conditional attributes into appropriate subclasses (e.g., HourlyEmployee)

  - E.g., Within a class, rather than passing values between methods through parameter lists, values are temporarily stored in object variables. These variables have meaningful values only when a particular method is running (undesirable)
    - <u>Replace Method with Method Object</u>

Long Method

- Long methods are hard to understand and more prone to bugs
- Find parts of the method that naturally go together, and Extract Method
- *Problem: How do the new sub-methods access the parameters and locals of the original method?*
- Store the original method's parameters and locals in instance variables so all sub-methods can access them?
  - No. This would cause the "Temporary Field" problem (fields that are not used all of the time)
- Parameters & locals of original method could be passed as parameters into sub-methods
  - Often works, but sometimes leads to long parameter lists on sub-methods
  - Could Introduce Parameter Object to shorten parameter lists

Long Method (cont.)

- If the long method is *really long*, or if the parameter lists on extracted sub-methods are too long, you can Replace Method with Method Object
  - Put the original method and all of its extracted sub-methods on a new class
  - Parameters & locals from the original method become instance variables on the new class, making them available to the extracted methods without passing parameters
  - Instantiate method object when you need to execute the method, then throw it away

Large Class

- Signs that a class is doing too much and needs further decomposition
  - Remove duplication using Extract Method
  - If there's still too much code, find groups of related methods and Extract Class or Extract Subclass
  - Doesn't use all of its instance variables all of the time
    - Find groups of instance variables that are only used some of the time, and Extract Subclass or Extract Class
  - Replace Method with Method Object

Long Parameter List

- Long parameter lists are hard to understand

- OO programming tends to make parameter lists shorter
  - Methods can get the data they need from the host class, or by calling methods on object parameters

- Refactorings for reducing the number of parameters
  - If the method can access a passed-in value in some other way, don't pass it in (Replace Parameter with Method)
  - If several parameters are related, Introduce Parameter Object to reduce the number of parameters
  - If callers extract multiple values from an object so they can be passed to a method, it might be easier to just pass in the whole object (Preserve Whole Object)
    - Unless you don't want to couple the two classes

Divergent Change

- Divergent types of changes require modifications to the same class
  - Class C must be modified when:
    - We change to a different data persistence technology
    - We change to a different UI implementation
    - We change to a different networking protocol
- Indicates the class is not cohesive (performs multiple unrelated responsibilities)
- Aspects of a system that are unrelated and will evolve independently should be implemented in different classes ("Separation of Concerns")
- Identify different areas of responsibility, and Extract Class to move each different responsibility to a new class

Shotgun Surgery

- Ideally,
  - Every design decision or policy is implemented at only one place in the code
  - Changing a design decision or policy requires modifying only one class (or a small number of classes)

- Example: We decide to move from MS SQL Server to Oracle database
- Example: We change our policy for handling database exceptions

- Shotgun Surgery - Making a particular kind of change requires making lots of little changes to many different classes
- Indicates a particular responsibility is spread throughout the system, and may need to be centralized in a single class

- Create one class to perform the responsibilities related to the change
  - Use Move Method and Move Field to move functionality to the new class
- Use Aspect-Oriented Programming (AOP)

Feature Envy

- A method on one class makes heavy use of the features on another class
  - Good OO design should package data together with the processes that use the data

- This is a sign that the method is on the wrong class
  - Move Method can fix that

- If only part of the method is "envious", use Extract Method to isolate the envious code, and use Move Method to move it to the other class

- What if the method uses data from several classes?
  - Put the method on the class that it's most intimate with
  - Use Extract Method to isolate the sections of code that interact heavily with other classes, and use Move Method to move the new methods where they belong

Data Clumps

- If multiple data items appear together in lots of places, it's likely that a class is missing

- Create a new class that encapsulates the data clump
- Consolidate behavior that's related to the data clump on the new class
  - Move Method
- Replace all occurrences of the data clump with instances of the new class
  - E.g., simplify parameter lists using Introduce Parameter Object

Primitive Obsession

- Some data items seem so simple that we use primitive data types to represent them
  - String name; String phoneNumber; int payGrade;
- Simple values like this tend to get more complicated over time
  - You need logic for parsing them, formatting them, changing them in controlled ways, etc.
  - Because the values are primitives, this logic is placed on other classes
  - This often leads to code duplication and feature envy
- Use Replace Data Value with Object to provide a proper home for this code

Switch Statements

- Switch statements are a form of duplication
  - Each switch hard-codes the list of cases
- Adding a new case requires changing all the switches
- Good OO design replaces switches on type codes with polymorphic method calls
  - Superclass defines a common interface containing dynamically-bound methods for all behaviors that vary between subclasses
  - Most code is written in terms of references to the superclass, and dynamically-bound method calls replace switch statements
  - New subclasses can be added without modifying existing code
    - We prefer to not touch code that already works

Switch Statements

- Use <u>Extract Method</u> to isolate switches on type codes
- Use <u>Move Method</u> to move new methods containing switches to the class containing the type code
- Use <u>Replace Conditional with Polymorphism</u> to get rid of switches
  - Set up inheritance hierarchy, and move the code from each switch case to the appropriate subclass

Parallel Inheritance Hierarchies

- You have two or more isomorphic inheritance hierarchies
- Whenever you add a class to one hierarchy, you also have to add corresponding classes to the other hierarchies

- Example: You might have inheritance hierarchies for
  - Domain objects
  - Data access objects
  - GUI editors
- Every time you add a new domain class, you also have to create a new data access class and a new editor class

- Results in <u>duplication</u> and <u>shotgun surgery</u>

Parallel Inheritance Hierarchies

- Solution? Collapse the parallel hierarchies into one hierarchy

- Example: One class represents the domain object, data access object, and GUI editor
  - A new concept can be added by creating only one class
  - But, we now have domain stuff, data access stuff, and GUI stuff combined on a single class
  - Is this really an improvement? How does it affect cohesion and layering?

- Often parallel hierarchies allow for better separation of concerns, and should be used (i.e., lesser of two evils)
- Sometimes it's better to collapse the hierarchies into one
- Code generation tools can help solve this problem. You still have parallel hierarchies, but only one must be maintained manually
  - E.g., Write tools that automatically generate the code for the data access and editor classes for a domain class

Speculative Generality

- "I think we need the ability to do this kind of thing someday, so let's build in support for it now"
- Speculating on future needs is a tricky business, so building a lot of infrastructure for features you may never need is dubious

- Signs of speculative generality:
  - Unused classes, methods, parameters
  - Complicated inheritance hierarchies that serve no current purpose
  - Levels of indirection that serve no current purpose

- Remove speculative generality by applying relevant refactorings
  - Remove Parameter, Inline Class, Collapse Hierarchy, Remove Middle Man, etc.

Message Chains

- `obj.getThat().getTheOther().getYetAnother().FinallyDoSomething()`
- The client is coupled to the structure of the navigation
  - If the intermediate object relationships change, so must the client
- Exposing delegates to clients is poor encapsulation

- Shorten the chain as much as possible
- Use Hide Delegate to hide any remaining delegates
  - `obj.FinallyDoSomething()`

Middle Man

- There are two options for reusing code from another class:
  - Inheritance: Inherit from the other class, thus acquiring its functionality
  - Composition: Create an instance of the other class and delegate method calls to it. The delegating class acts as a "middle man"

- Inheritance is easier because any changes made to the superclass are automatically inherited by the subclass
- Composition allows control over which of the other class' features are exposed by the client class, but requires work to write the delegating methods

- If a middle man does a lot of simple delegation to another class, consider the following refactorings
  - <u>Remove Middle Man</u>: provide accessor for delegate so that clients can call it directly (could be harmful to encapsulation)
  - <u>Replace Delegation with Inheritance</u> to avoid the work necessary to write the delegating methods (but all features of the superclass will be exposed)

Inappropriate Intimacy

- Classes know too much about each other
- "Classes should follow strict, puritan rules"
- Hide implementation details behind a minimal public interface
- "Fragile Base Class" problem
  - Subclasses depend on internal details of a superclass. Changes to the superclass break the subclasses
  - Internal details should be hidden even from subclasses (private is better than protected)
  - Replace Inheritance with Delegation

Alternative Classes with Different Interfaces

- Two classes have methods that do similar things, but they use different naming conventions
  - Delete vs. Remove
  - Initialize vs. Setup
- People create similar code to handle similar situations, but don't realize the other code exists (i.e., duplication)

- Use <u>Rename Method</u>, <u>Add Parameter</u>, <u>Remove Parameter</u>, etc. to make the two sets of methods consistent

- If the classes can be modified to share code, use <u>Extract Class</u>, <u>Extract Method</u>, etc. to remove duplication

Incomplete Library Class

- A library class lacks some needed functionality, but we can't refactor the class because we didn't write it, don't have the code, etc.

- Introduce Local Extension
  - Make a subclass of the library class that has the additional functionality
  - If the library class can't be subclassed (i.e., it's "final"), or you don't control creation of the objects, you'll have to use a wrapper instead of a subclass

- If your language supports it, write an “extension method” to extend the library class without subclassing or wrapping it (C# and Objective-C support this)

- Introduce Foreign Method
  - Create a method on the client class with an instance of the library class as the first argument
    - private static Date nextDay(Date arg) { … }
  - Works if only a few methods need to be added

Data Class

- A class containing only fields and possibly getters/setters for those fields
  - a.k.a. "structure" or "dumb data holder"
- Data classes are often manipulated in too much detail by other classes
  - Feature envy is common when data classes are used

- Use Encapsulate Field to encapsulate public fields
- Use Encapsulate Collection to encapsulate collection fields
- Use Remove Setting Method to protect read-only fields
- Look at what other classes are doing with the data class, and use Move Method to reduce feature envy
- If you can't move entire methods, use Extract Method first to isolate the envious code, and then move it to the data class using Move Method

Lazy Class

- Effective OO design often leads to lots of classes
  - But, each class costs money to understand and maintain
- A good design has enough classes to fully decompose the system into cohesive units, but no more
  - Too few classes is bad. So is too many.

- Lazy Class: A class that isn't doing enough to justify it's existence
  - Prior functionality has been moved to other classes (Move Field, Move Method, etc.)
  - You had plans for the class that never materialized
- Get rid of lazy classes
  - Use Inline Class to move its functionality to another class
    - Fold TelephoneNumber class into Employee class?
  - Use Collapse Hierarchy to move its functionality to its superclass
    - Collapse PartTimeStudent into Student superclass?

Refused Bequest

- A subclass wants to inherit only part of its superclass's functionality
- In order to disable unwanted functionality, the sublass overrides unwanted methods to throw UnsupportedOperationException or just "do nothing"
- The subclass doesn't fully support the superclass's interface, and so isn't really a subtype (i.e., subclass objects can't be substituted in place of the superclass)

- Solution 1
  – Use Replace Inheritance with Composition to allow reuse without establishing a subtyping relationship
- Solution 2
  – Create a new sibling class and use Push Down Method and Push Down Field to push all unwanted functionality into the sibling

Comments

- Comments are good, but sometimes they're used as an excuse for writing bad code

- Before commenting some code, ask if there is a way to write it more clearly so it doesn't need comments

- If you feel the need to comment a block of code, use Extract Method to move the code into its own method
- Pick a good name so it's clear what the method does (use Rename Method until you get it right)

- If a comment makes a statement about the program's state at a particular point, use Introduce Assertion to replace the comment with an assertion