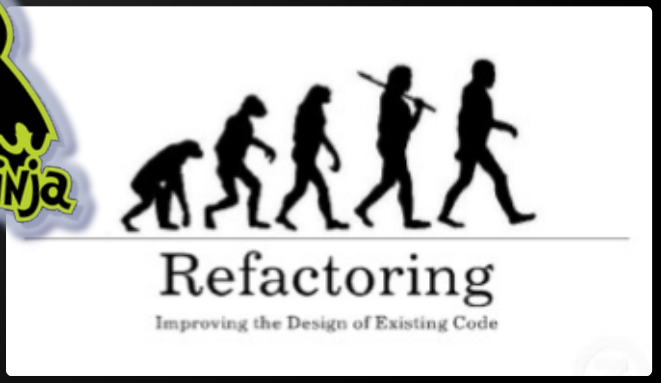
Telerik Academy

Refactoring: Improving the Quality of Existing Code

When and How to Refactor? Refactoring Patterns



Telerik Software Academy
Learning & Development
http://academy.telerik.com

Telerik Academy

Table of Contents

- What is Refactoring?
- Refactoring principles
- Refactoring process and tips
- Code smells
- Refactorings
  - Data level, statement level, method level, class level, system level refactorings, etc.
- Refactoring patterns

What is Refactoring?

Refactoring means "to improve the design and quality of existing source code without changing its external behavior".

Martin Fowler

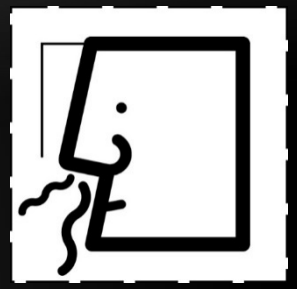
- It is a step by step process that turns the bad code into good code
  - Based on "refactoring patterns" → well-known recipes for improving the code

Code Refactoring

- What is refactoring of the source code?
  - Improving the design and quality of existing source code without changing its behavior
  - Step by step process that turns the bad code into good code (if possible)
- Why we need refactoring?
  - Code constantly changes and its quality constantly degrades (unless refactored)
  - Requirements often change and code needs to be changed to follow them

When to Refactor?

- Bad smells in the code indicate need of refactoring



- Refactor:
  - To make adding a new function easier
  - As part of the process of fixing bugs
  - When reviewing someone else's code
  - Have technical debt (or any problematic code)
  - When doing test-driven development
- Unit tests guarantee that refactoring does not change the behavior
  - If there are no unit tests, write them

Refactoring Main Principles

- Keep it simple
- Avoid duplication (DRY)
- Make it expressive (self-documenting, comments, etc.)
- Reduce overall code
- Separate concerns
- Appropriate level of abstraction
- Boy scout rule
  - Leave your code better than you found it

Refactoring Process

- Save the code you start with
  - Check-in or backup the current code
- Make sure you have tests to assure the behavior after the code is refactored
  - Unit tests / characterization tests
- Do refactorings one at a time
  - Keep refactorings small
  - Don't underestimate small changes
- Run the tests and they should pass / else revert
- Check-in

Refactoring Tips

- Keep refactorings small
- One at a time
- Make a checklist
- Make a "later"/TODO list
- Check-in/commit frequently
- Add tests cases
- Review the results
  - Pair programming
- Use tools (Visual Studio + Add-ins)

Code Smells

Code Smells

- Certain structures in the code that suggest the possibility of refactoring
- Types of code smells
  - The bloaters
  - The obfuscators
  - Object-oriented abusers
  - Change preventers
  - Dispensables
  - The couplers

Code Smells: The Bloaters

- Long method
  - Small methods are always better (easy naming, understanding, less duplicate code)
- Large class
  - Too many instance variables or methods
  - Violating Single Responsibility principle
- Primitive obsession (Overused primitives)
  - Over-use of primitives, instead of better abstraction
  - Part of them can be extracted in separate class and encapsulate their validation there

Code Smells: The Bloaters (2)

- Long parameter list (in/out/ref parameters)
  - May indicate procedural rather than OO style
  - May be the method is doing too much things
- Data clumps
  - A set of data items that are always used together, but are not organized together
  - E.g. credit card fields in order class

- Combinatorial explosion
  - Ex. ListCars, ListByRegion, ListByManufacturer, ListByManufacturerAndRegion, etc.
  - Solution may be Interpreter (LINQ)

Code Smells: The Bloaters (3)

- Oddball solution
  - A different way of solving a common problem
  - Not using consistency
  - Solution: Substitute algorithm or use adapter
- Class doesn't do much
  - Solution: Merge with another class or remove
- Required setup/teardown code
  - Requires several lines of code before its use
  - Solution: Use parameter object, factory method, IDisposable

Telerik Academy

Code Smells: The Obfuscators

- Regions
  - The intend of the code is unclear and needs commenting (smell)
  - The code is too long to understand (smell)
  - Solution: partial class, a new class, organize code
- Comments
  - Should be used to tell WHY, not WHAT or HOW
  - Good comments: provide additional information, link to issues, explain reasons, give context
  - Link: Funny comments

Telerik Academy Code Smells: The Obfuscators (2)

- Poor/improper names
  - Should be proper, descriptive and consistent
- Vertical separation
  - You should define variables just before first use
    - Avoid scrolling
- Inconsistency
  - Follow the POLA
  - Inconsistency is confusing and distracting
- Obscured intent
  - Code should be as expressive as possible

Code Smells: OO Abusers

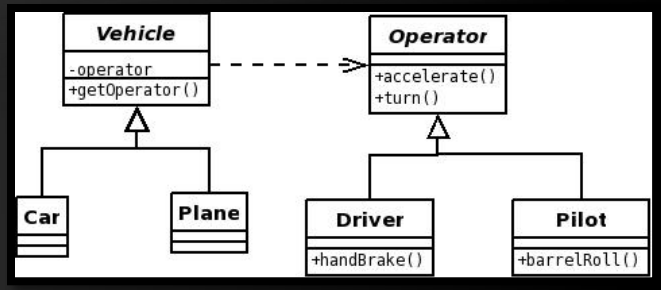
- Switch statement
  - Can be replaced with polymorphism
- Temporary field
  - When passing data between methods
- Class depends on subclass
  - The classes cannot be separated (circular dependency)
  - May broke Liskov substitution principle
- Inappropriate static
  - Strong coupling between static and callers
  - Static things cannot be replaced or reused

Code Smells: Change Preventers

- Divergent change
  - A class is commonly changed in different ways for different reasons
  - Violates SRP (single responsibility principle)
  - Solution: extract class
- Shotgun surgery
  - One change requires changes in many classes
    - Hard to find them, easy to miss some
  - Solution: move method, move fields
  - Ideally there should be one-to-one relationship between changes and classes

Telerik Academy Code Smells: Change Preventers

- Parallel inheritance hierarchies
  - New vehicle = new operator
  - Frequently share same prefix
  - Hard to be completely avoided. We can merge the classes or use the Intelligent children pattern



- Inconsistent abstraction level
  - E.g. code in a method should be one level of abstraction below the method's name

Telerik Academy Code Smells: Change Preventers

- Conditional complexity
  - Cyclomatic complexity (number of unique paths that the code can be evaluated)
  - Symptoms: deep nesting (arrow code) & bug ifs
  - Solutions: extract method, strategy pattern, state pattern, decorator
- Poorly written tests
  - Badly written tests can prevent change
  - Tight coupling

Code Smells: Dispensables

- Lazy class
  - Classes that don't do enough to justify their existence should be removed
  - Every class costs something to be understand and maintained
- Data class
  - Some classes with only fields and properties
  - Missing validation? Class logic split into other classes?
  - Solution: move related logic into the class

Code Smells: Dispensables (2)

- Duplicated code
  - Violates the DRY principle
  - Result of copy-pasted code
  - Solutions: extract method, extract class, pull-up method, template method pattern
- Dead code (code that is never used)
  - Usually detected by static analysis tools
- Speculative generality
  - "Some day we might need…"
  - YAGNI principle

Code Smells: The Couplers

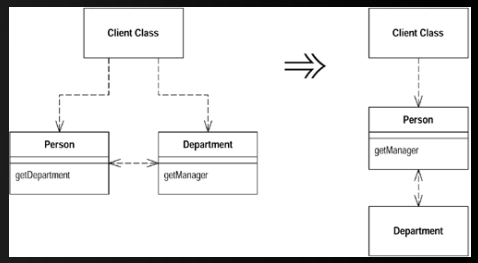
- Feature envy
  - Method that seems more interested in a class other than the one it actually is in
  - Keep together things that change together
- Inappropriate intimacy
  - Classes that know too much about one another
  - Smells: inheritance, bidirectional relationships
  - Solutions: move method/field, extract class, change bidirectional to unidirectional association, replace inheritance with delegation

Code Smells: The Couplers (2)

- The Law of Demeter (LoD)
  - A given object should assume as little as possible about the structure or properties of anything else
  - Bad e.g.: customer.Wallet.RemoveMoney()
- Indecent exposure
  - Some classes or members are public but shouldn't be
  - Violates encapsulation
  - Can lead to inappropriate intimacy

Code Smells: The Couplers (3)

- Message chains
  - Somemthing.another.someother.other.another
  - Tight coupling between client and the structure of the navigation



- Middle man
  - Sometimes delegation goes too far
  - Sometimes we can remove it or inline it

- Tramp data
  - Pass data only because something else need it
  - Solutions: Remove middle man, extract class

Code Smells: The Couplers (4)

- Artificial coupling
  - Things that don't depend upon each other should not be artificially coupled
- Hidden temporal coupling
  - Operations consecutively should not be guessed
  - E.g. pizza class should not know the steps of making pizza -> template method pattern
- Hidden dependencies
  - Classes should declare their dependencies in their constructor
  - "new" is glue / Dependency inversion principle

Refactorings

Data Level Refactorings

- Replace a magic number with a named constant
- Rename a variable with more informative name
- Replace an expression with a method
  - To simplify it or avoid code duplication
- Move an expression inline
- Introduce an intermediate variable
  - Introduce explaining variable
- Convert a multi-use variable to a multiple single-use variables
  - Create separate variable for each usage

Data Level Refactorings (2)

- Create a local variable for local purposes rather than a parameter
- Convert a data primitive to a class
  - Additional behavior / validation logic (money)
- Convert a set of type codes (constants) to enum
- Convert a set of type codes to a class with subclasses with different behavior
- Change an array to an object
  - When you use an array with different types in it
- Encapsulate a collection

Statement Level Refactorings

- Decompose a boolean expression
- Move a complex boolean expression into a well-named boolean function
- Consolidate duplicated code in conditionals
- Return as soon as you know the answer instead of assigning a return value
- Use break or return instead of a loop control variable

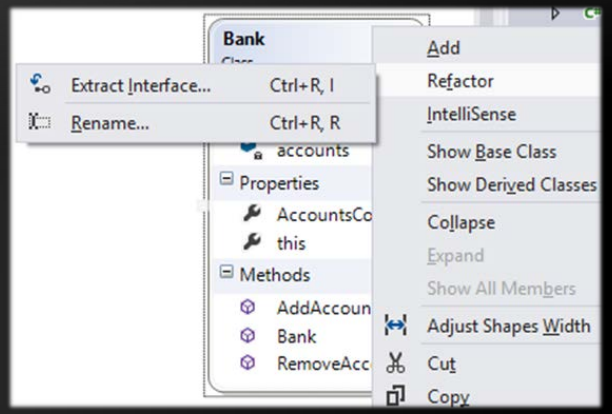
- Replace conditionals with polymorphism
- Use null objects instead of testing for nulls

Method Level Refactorings

- Extract method / Inline method
- Rename method
- Convert a long routine to a class
- Add / remove parameter
- Combine similar methods by parameterizing them
- Substitute a complex algorithm with simpler
- Separate methods whose behavior depends on parameters passed in (create new ones)
- Pass a whole object rather than specific fields
- Encapsulate downcast / Return interface types

Class Level Refactorings

- Change structure to class and vice versa
- Pull members up / push members down the hierarchy



- Extract specialized code into a subclass
- Combine similar code into a superclass
- Collapse hierarchy
- Replace inheritance with delegation
- Replace delegation with inheritance

Class Interface Refactorings

- Extract interface(s) / Keep interface segregation
- Move a method to another class
- Convert a class to two
- Delete a class
- Hide a delegating class (A calls B and C when A should call B and B call C)
- Remove the man in the middle
- Introduce (use) an extension class
  - When you have no access to the original class
  - Alternatively use decorator pattern

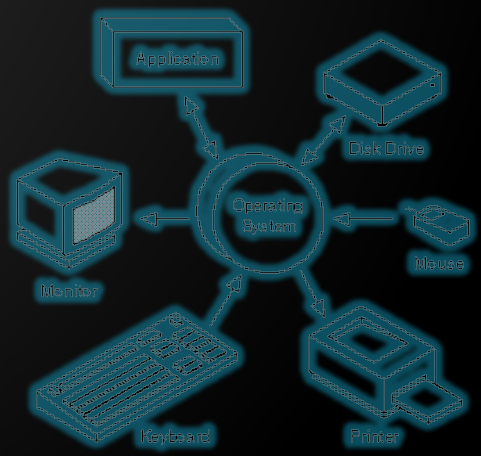
Telerik Academy

Class Interface Refactorings (2)

- Encapsulate an exposed member variable
  - In C# always use properties
  - Define proper access to getters and setters
    - Remove setters to read-only data
- Hide data and routines that are not intended to be used outside of the class / hierarchy
  - private -> protected -> internal -> public
- Use strategy to avoid big class hierarchies
- Apply other design patterns to solve common class and class hierarchy problems (façade, adapter, etc.)

System Level Refactorings

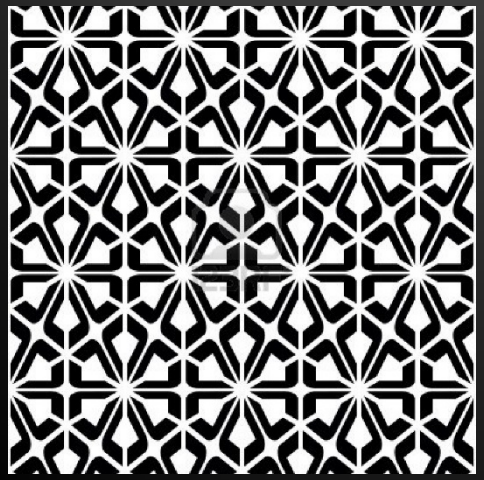
- Move class (set of classes) to another namespace / assembly
- Provide a factory method instead of a simple constructor / Use fluent API
- Replace error codes with exceptions
- Extract strings to resource files
- Use dependency injection
- Apply architecture patterns

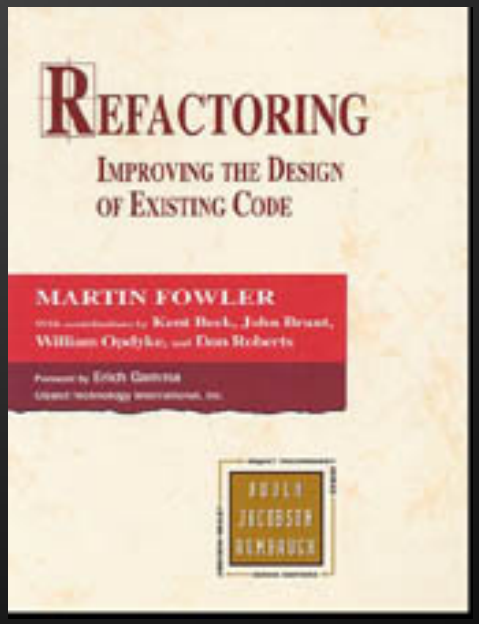


Telerik Academy

Refactoring Patterns

Well-Known Recipes for Improving the Code Quality





Rafactoring Patterns

- When should we perform refactoring of the code?
  - Bad smells in the code indicate need of refactoring
- Unit tests guarantee that refactoring does not change the behavior
- Rafactoring patterns
  - Large repeating code fragments → extract repeating code in separate method
  - Large methods → split them logically
  - Large loop body or deep nesting → extract method

Refactoring Patterns (2)

- Refactoring patterns
  - Class or method has weak cohesion → split into several classes / methods
  - Single change carry out changes in several classes → classes have tight coupling → consider redesign
  - Related data are always used together but are not part of a single class → group them in a class
  - A method has too many parameters → create a class to groups parameters together
  - A method calls more methods from another class than from its own class → move it

Rafactoring Patterns (3)

- Refactoring patterns
  - Two classes are tightly coupled → merge them or redesign them to separate their responsibilities
  - Public non-constant fields → make them private and define accessing properties
  - Magic numbers in the code → consider extracting constants
  - Bad named class / method / variable → rename it
  - Complex boolean condition → split it to several expressions or method calls

Rafactoring Patterns (4)

- Refactoring patterns
  - Complex expression → split it into few simple parts
  - A set of constants is used as enumeration → convert it to enumeration
  - Method logic is too complex and is hard to understand → extract several more simple methods or even create a new class
  - Unused classes, methods, parameters, variables → remove them
  - Large data is passed by value without a good reason → pass it by reference

Rafactoring Patterns (5)

- Refactoring patterns
  - Few classes share repeating functionality → extract base class and reuse the common code
  - Different classes need to be instantiated depending on configuration setting → use factory
  - Code is not well formatted → reformat it
  - Too many classes in a single namespace → split classes logically into more namespaces
  - Unused `using` definitions → remove them
  - Non-descriptive error messages → improve them
  - Absence of defensive programming → add it

Telerik Academy

Code Refactoring

http://academy.telerik.com

Homework

- Refactor the C# code from the Visual Studio Project "Refactoring-Homework.zip" to improve its internal quality. You might follow the following steps:
  1. Make some initial refactorings like:
     - Reformat the code.
     - Rename the ugly named variables.
  2. Make the code testable.
     - Think how to test console-based input / output.
  3. Write unit tests. Fix any bugs found in the mean time.
  4. Refactor the code following the guidelines from this chapter. Do it step by step. Run the unit tests after each major change.

Telerik Academy

Free Trainings @ Telerik Academy

- C# Programming @ Telerik Academy
  - csharpfundamentals.telerik.com

- Telerik Software Academy
  - academy.telerik.com

- Telerik Academy @ Facebook
  - facebook.com/TelerikAcademy

- Telerik Software Academy Forums
  - forums.academy.telerik.com