Principles in Refactoring



(chapter 2)

Refactoring is

changing

the internal structure of software

keeping

the same observable behavior



Goal

• The goal is to make the software easier to understand (clean up code, reduce complexity) without noticeable behavior changes.

 Performance optimization, instead, is not a goal!

Writing Tests

- Do not add any new tests, only restructure the code
- Only change existing tests to reflect occurred interface changes



The Two Hats

- Add new functions first
- Then refactor them considering the previous existing code



Regular refactoring helps

- Improving the design of software
- Tiding up the code, removing duplications
- Finding bugs
- Writing robust code
- Programming faster



When to refactor

- After writing duplicated code
- After adding new functions
- On fixing bugs
- On code reviewing



Management

- Non-technical managers are not concerned by refactoring, thus hard to accept and schedule them.
- In any case during the development process, refactoring has to be performed



Indirection

- Breaking big objects and big methods in smaller parts, reducing the complexity:
 - Sharing logic (i.e. helper method, superclass)
 - Explain intention (i.e. method name)
 - Isolate change
 - Encode conditional logic
- But it introduces the need of manage more objects and functions
 - Ensure then to avoid parasitic indirection



Problems with Refactoring

- Databases coupling
- Changing (published) interface
- Introducing exception throws
- Code is too messy and buggy (candidate to rewrite)
- Unfinished refactoring at the deadline

Design

- Do not try to have the perfect design on the first run
- Re-design over refactoring, better understand the problem to solve
- Flexibility never needed leads to overheads
- Simple solution first, then refactor for needed flexibility



Bad Smells in Code

- Duplicated code
 - → Dry! Reuse helper method
- Long method
 - → Decompose in smaller methods
- Large class
 - → Decompose in subclasses
- Long parameter list
 - → Only pass what always required, call methods to get additional needed data
 - → Pass map, keeping unchanged the interface

- Divergent change
 - → Separate classes to reduce dependencies on change
- Shotgun surgery
 - Collect methods depending on common change
- Feature envy
 - → Implement methods within the responsible classes

Refactoring Techniques

(chapter 3)

- Extract Method
 - → Short and finely grained methods
 - → Close semantic purpose
 - → Self explaining code, no need of comments
 - → Well-named methods explaining the specific functionality
 - → Easy to override
- Pull Up Method



→ Eliminate duplicated behavior from classes, preventing partial alterations

Refactoring Techniques

- Form Template Method
 - → Reduce duplicated behavior on subclasses
 - → Exploit polymorphism
- Substitute Algorithm
 - → Replace complex method body with simpler one
 - → Easy to understand
 - → Avoid duplication with library features
- Extract Class
 - → Short and well defined purpose
 - → Split classes grouping common responsibilities

Refactoring Techniques



- Replace Temp With Query
 - → Prevent using local variable storing expression result
 - → Accessibility of an expression, extracting it into a method
 - → Possible reuse of the new method
- Introduce Parameter Object
 - → Reduce list of parameters
 - → Group relative parameters into object
 - → Detect potential extract class candidates



Preserve Whole Object

- → Reduce parameter list
- → Coupling direct source object instead of call many methods for its data
- Cons: increase dependencies between objects
- Replace Method With Method Object
 - → A method with many local variables is a candidate to be converted to an object with its attributes
 - → Complex method can still be extracted within the new class

Decompose Conditionals

- → Reduce complexity of control flow
- → Better describing of code branching
- → Accessibility of conditional checks, extracting them into methods
- → Possible reuse of the new methods
- Extract Subclass
 - → Separate methods only used by some instances of a class
 - → Better specification of a class

- Replace Data Value with Object
 - → Transform object attribute from native value to a separate entity
 - → Relational object
- Replace Type Code with Class
 - → Rely on symbolic names instead of numeric type codes
 - → Enforce compiler check



- Replace Type Code with Subclasses
 - → Split behavior depending the object type
 - → Exploit polymorphism
- Replace Type Code with State/Strategy
 - → Split behavior and switch them during the life cycle of the instance
- Replace Array with Object
 - → Semantic description of the values

Bad Smells in Code

- Data Clumps
 - → Extract group of data items into value object
- Primitive Obsession
 - → Encapsulate primitive type attributes into value object
- Switch Statements
 - → Exploit polymorphism
- Parallel Inheritance Hierarchies
 - → Consider folding the hierarchy into a single class



Bad Smells in Code

- Lazy Class
 - → Eliminate classes without enough responsibilities
 - → Collapse hierarchy if any
- Speculative Generality
 - → Avoid implementation of non-required and never used features
- Temporary Field
 - → Ensure instance variables are always used
 - → Extract instance variables and relative algorithms in separate class
- Message Chains
 - → Method call delegation through many related objects could break the chain

