Software Development

Refactoring & Clean Code

Koen Pelsmaekers

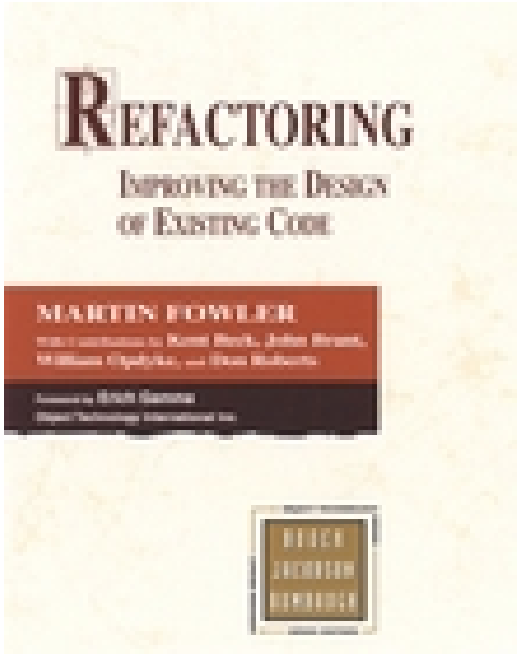
Unit Informatie (GT 03.14.05)

email: koen.pelsmaekers@kuleuven.be

Android app!

The Bible of Refactoring

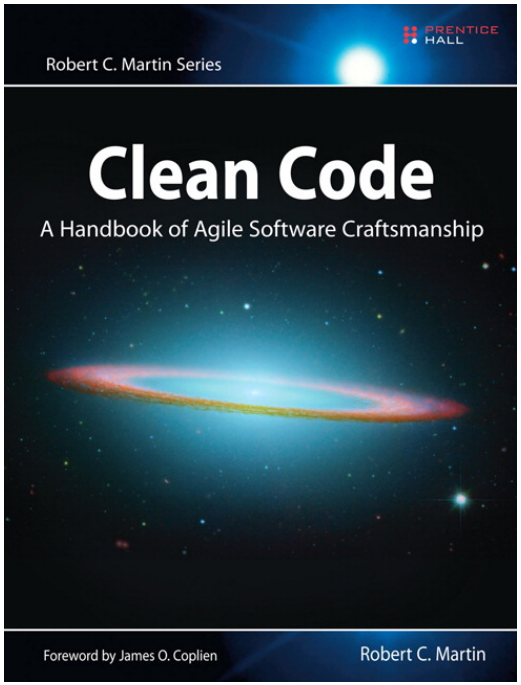
- Refactoring, Improving the Design of Existing Code, Martin Fowler (Kent Beck).



"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."

The Bible of Writing Good Code

- Clean Code, A Handbook of Agile Software Craftmanship, Robert "uncle Bob" C. Martin.



"@author, we are authors"

"code is written to be read"

"ratio of time spent reading vs. writing is 10:1"

Good design principle:

Program to an interface

- Decouple declaration from implementation
  - "What" vs. "How"
- Information hiding or Encapsulation
  - Do not expose the internals of your implementation
- Defer choice of actual class

- Criteria for designing a good interface
  - **Cohesion**: implements a single abstraction
  - **Completeness**: provides all operations necessary
  - **Convenience**: makes common tasks simple
  - **Clarity**: do not confuse your programmers
  - **Consistency:** keep the level of abstraction

What does "Clean code" mean?

- Meaningful names for
  - variables, functions, arguments, classes, packages, ...
- Formatting code
  - indentation, distance: use IDE settings, be consistent
- Comments
  - "Don't comment bad code – rewrite it." (Kernighan)
- Methods/functions
  - small, do one thing, one level of abstraction, as few arguments as possible, no side effects, DRY [*] (= no duplicate code)
- Information hiding
  - do not expose the internal implementation
- (Automatic) unit tests
  - be sure your code works and make it easy to test, test, test...

[*] DRY = Don’t Repeat Yourself

What does "Clean code" mean?

- No side effects
  - A function "promises" to do one thing (and returns a value), but it also does other *hidden* things
    - Unexpectedly changes to fields, to parameters or global variables
    - Temporal couplings; order of evaluation matters
    - Perform I/O
  - Avoid side effects by introducing functional programming
    - Output of one functions is passed to the next, without changing the content of other variables

What is "refactoring"?

- Fowler (noun): www.refactoring.com
  - "A disciplined technique for restructuring an existing body of code, alterning its internal structure without changing its external behavior"
- Fowler (verb)
  - "To restructure software by applying a series of refactorings without changing its observable behavior"
- Roberts
  - "A behaviour-preserving source-to-source program transformation"
- Beck
  - "A change to the system that leaves its behaviour unchanged, but enhances some non-functional quality - simplicity, flexibility, understandability, ..."

Why do we need refactoring?

- To improve the software design
- To reduce
  - software decay/software aging
  - software complexity
  - software maintenance cost
- To increase
  - Software understandability (readability)
    - f.i. by introducing design patterns
  - Software productivity
    - at long term, not a short term
- To facilitate future changes
  - Useful in agile development
- ...

When do we refactor?

- Consolidation phase in iterative development
  - adding new functionality – consolidate
  - especially when the functionality is difficult to integrate in the existing code base
- When you think it is necessary
  - not on a periodical basis
- Apply the rule of three
  - first time: implement solution from scratch
  - second time: implement something similar by duplicating code
  - third time: do not reimplement or duplicate, but factorise!
- During normal code inspections/reviews

Some hints...

- It is no performance optimization
- Never refactor without unittest!
- Refactor in small steps
- Use IDE refactoring & Code inspection support
- Identify *bad smells* in source code (cf. next slide)
- Some examples
  - rename a class, extract a method, change the signature of a method, ... or more complex operations as introduce inheritance instead of switch/instanceof
- Available for all major OO languages (Java, C++, C#, ...), all major operating systems, all major IDE's

Java & performance

- Java compiler (javac)
  - Compilation to platform-neutral byte code (.class) at compile time
  - Hardly any optimizations (f.i. expression evaluation, ...)
  - Demo: decompilation
- Java Virtual Machine (JVM)
  - Interpretes byte code
  - JIT-compiler: Just In Time compilation from byte code to native machine code at runtime of most heavily used methods
  - Different levels of optimization
    - Cold, Warm, Hot (> 1% of time), Very hot, Scorching (> 12.5% of time)
    - Higher level, better performance, more resources (memory/CPU)

When to refactor? "Bad smells in code"

- Duplicated Code
  - Extract Method
  - Pull up Method
  - Extract Class
  - Form Template Method
- Long method
  (the longer the method the harder it is to see what it is doing)
  - "every comment"
  - Extract Method
  - Replace temp with Query
  - Introduce Parameter Object
  - Preserve Whole Object

When to refactor? (cont.)

- Large Class: too much responsibility?
  - "too many fields"
    - Extract Class
    - Extract Subclass
  - "too much code"
    - Extract Class
    - Extract Subclass
    - Extract Interface
- Long Parameter List
  - Replace Parameter with Method
  - Preserve Whole Object
  - Introduce Parameter Object

When to refactor? (cont.)

- Data Clumps
  - "data items enjoy hanging around in groups together"
  - Extract Class
  - Introduce Parameter Object
  - Preserve Whole Object
- Switch/case or if Statements & Typecasts
  - Polymorphism & dynamic binding; replace "procedural" code by "object-oriented" code
- Data Class
  - "class with few methods (only getters/setters)
  - Encapsulate Field
  - Encapsulate Collection
  - Move Method (try to move behavior into the class)
- Comments (to camouflage bad code [*])
  - Extract Method

[*] debug your code, not your comment.