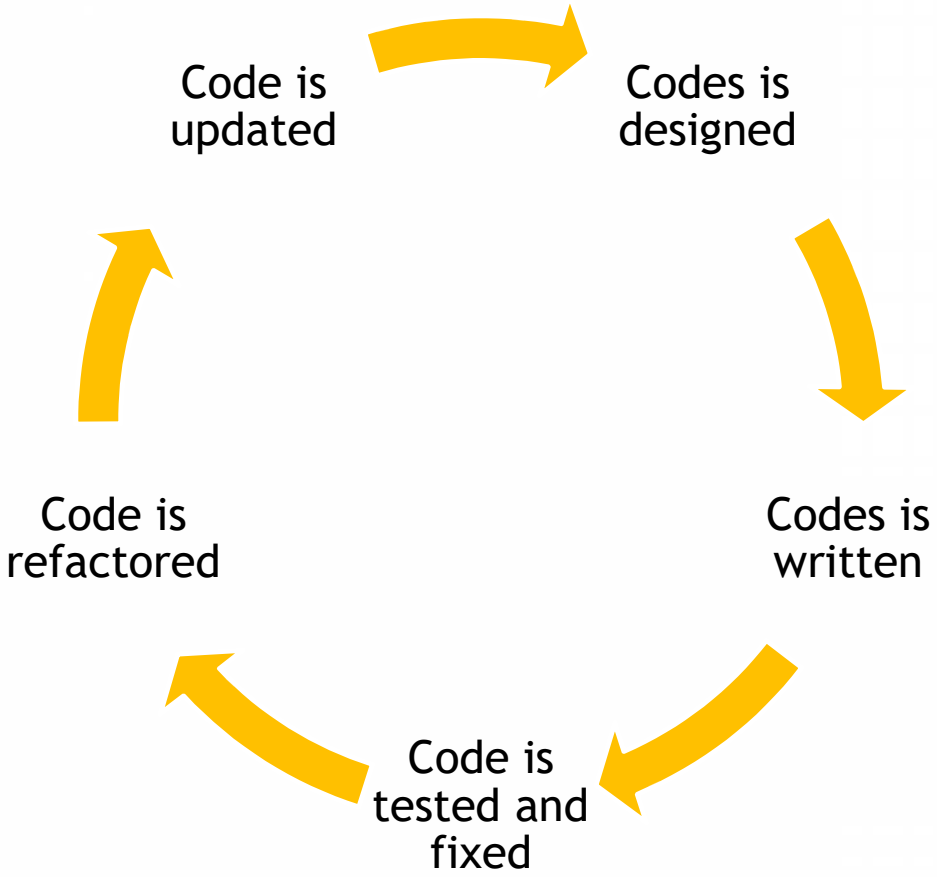
Codes Improvement

Ridi Ferdiana
ridi@acm.org
Version 1.0.0

The Codes Lifecycle



Refactoring

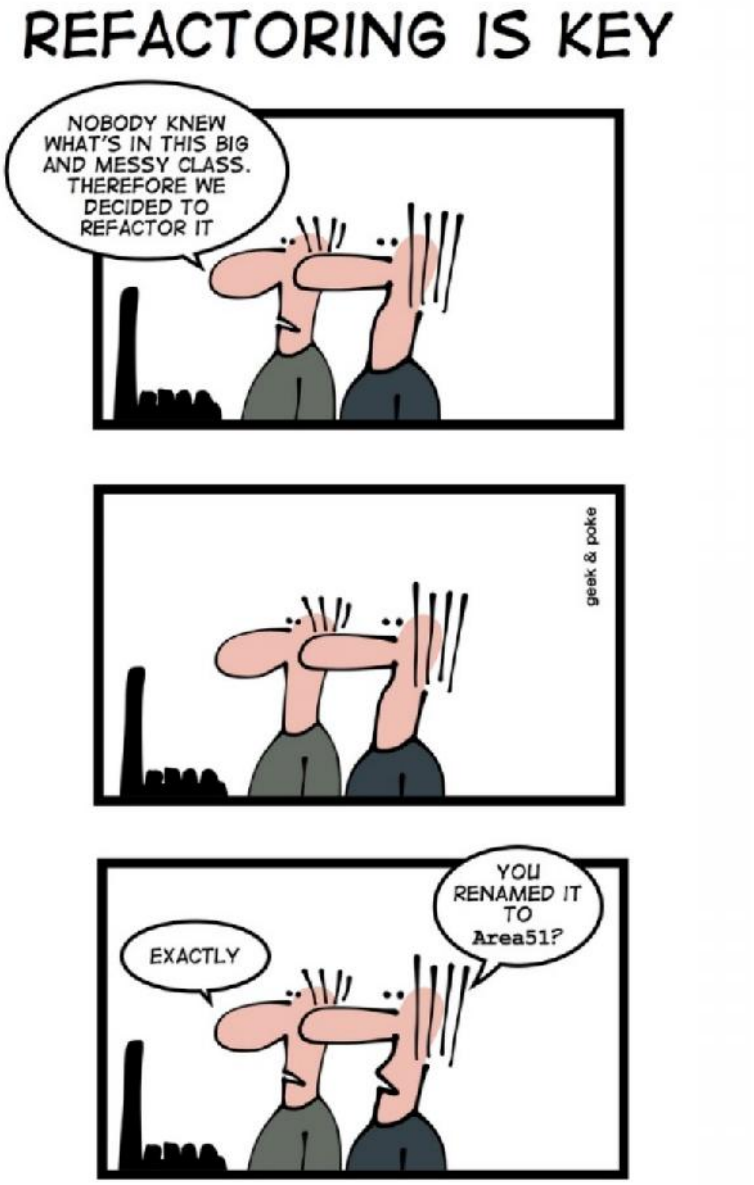
- Refactoring is about improving the codes

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

Martin Fowler

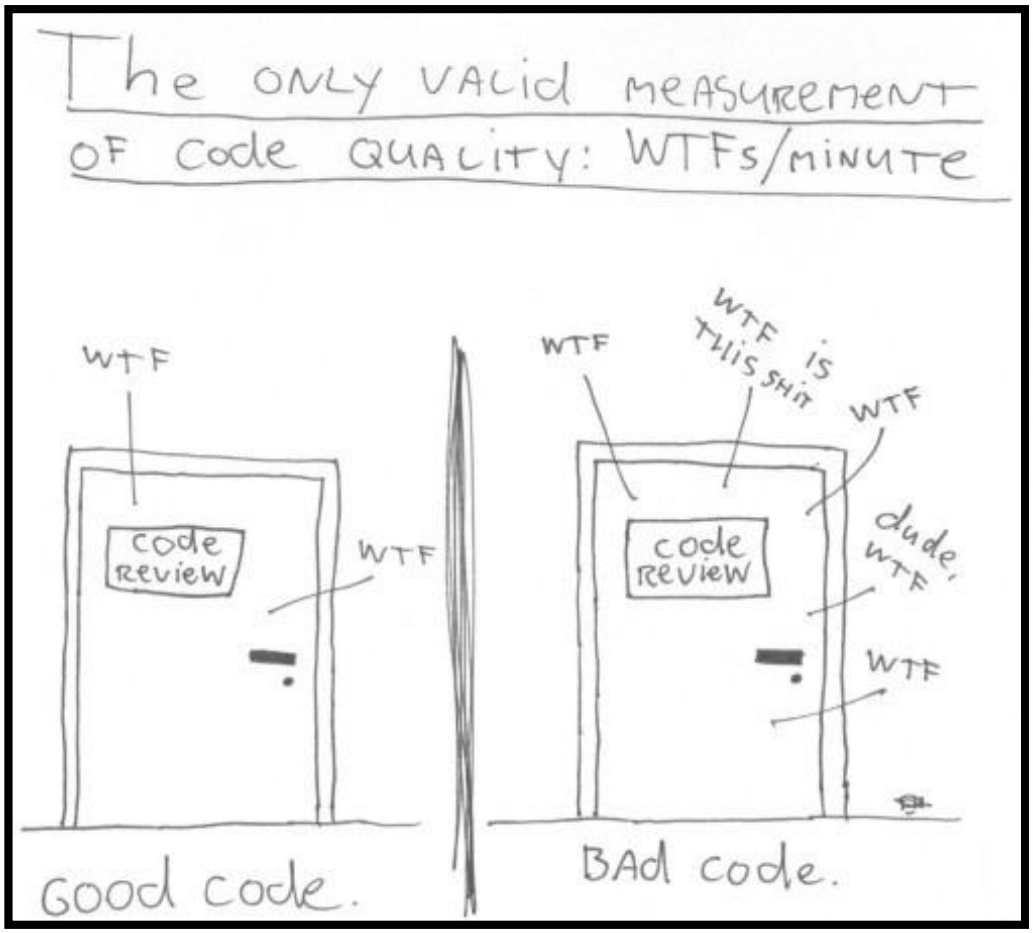
Why Codes Refactoring

- 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 small codes
- After fixing the bugs
- Reviewing others codes
- Test Driven Development

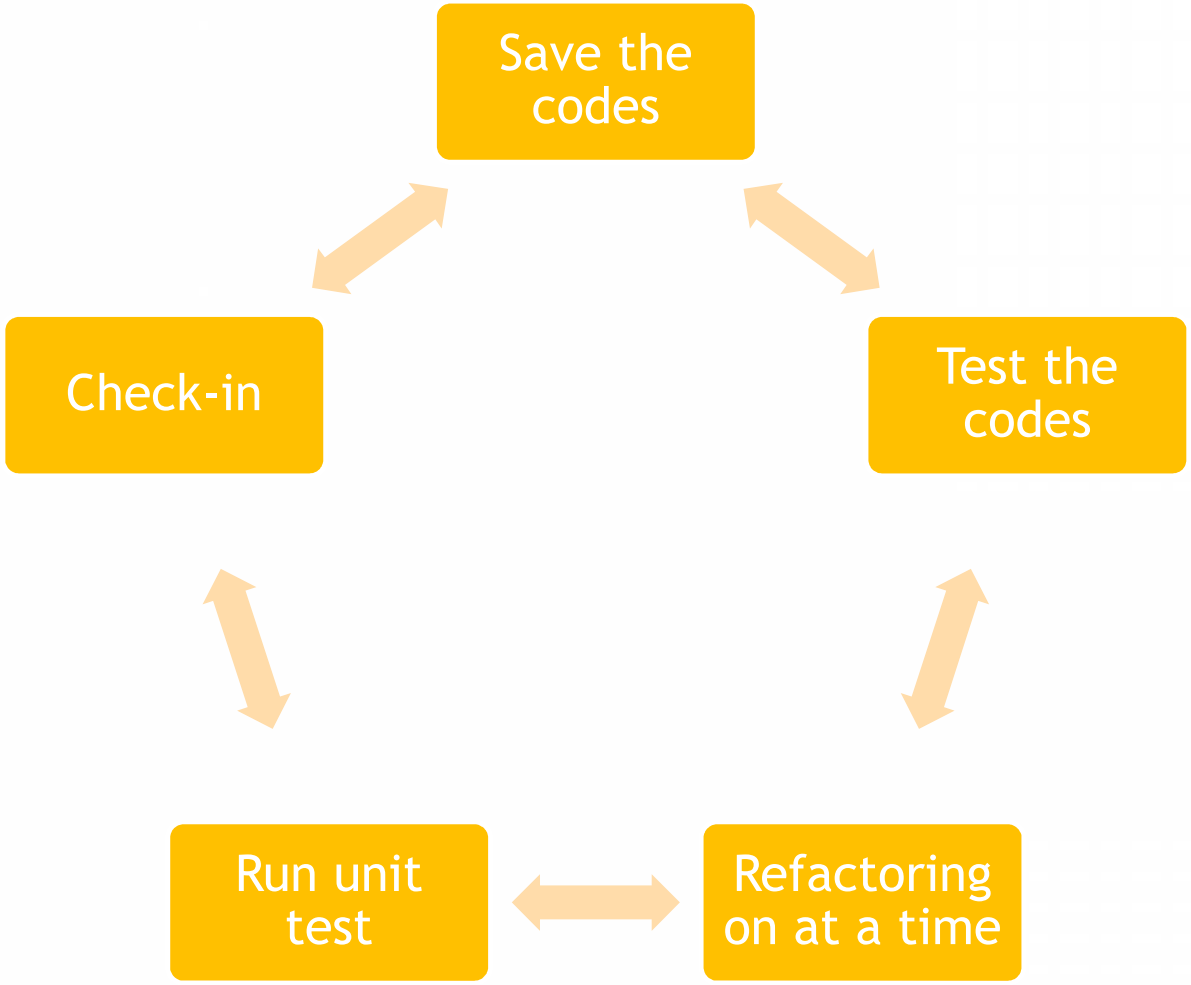


WTF = ‘What The Funny’ Codes
SHIT = ‘So Heavy in Test”

Good Codes Main Principles

- Avoid duplication (DRY)
- Simplicity – Keep it simple smart (KISS)
- Make it expressive (self-documenting, comments)
- Reduce overall code (YAGNI)
- More code = more bugs
- Avoid premature optimization
- Appropriate level of abstraction
- Hide implementation details

Refactoring Process



Code Smells : The Bloaters

- Long method
- Large class
- Primitive obsession (overused primitives)
  - Over-use of primitives, instead of better abstraction
- Long parameter list (in/out/ref parameters)
- Data clumps
  - A set of data items that are always used together
- Combinatorial explosion
  - ListCars, ListByRegion, ListByManufacturer, ListByManufacturerAndRegion

Code Smells : The Bloaters

- 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

Code Smells: Obfuscator

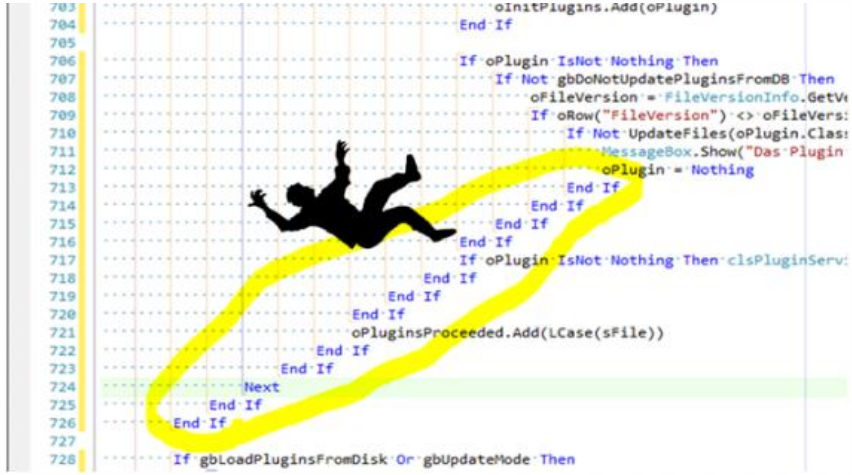
- Poor/improper names
  - Should be proper, descriptive and consistent
- Vertical separation
  - You should define variables just before first use
- Inconsistency
  - Inconsistency is confusing and distracting
- Obscured intent
  - Code should be as expressive as possible

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
- Shotgun surgery
  - One change requires changes in many classes
- Conditional complexity
  - Symptoms: deep nesting (arrow code) & bug ifs



Code Smells: Dispensables

- Lazy Class
  - Classes that don't do enough to justify their existence should be removed
- Data class
  - Some classes with only fields and properties
  - Missing validation? Class logic split into other classes?
- Duplicated codes
- Dead Codes
- Speculative Codes
  - "Some day we might need…"

Code Smells: Couplers

- Inappropriate intimacy
  - Method that seems more interested in a class other than the one it actually is in
- Feature envy
  - Classes that know too much about one another
- Indecent exposure
  - Some classes or members are public but shouldn't be
- The Law of Demeter (LoD
  - Least knowledge Bad e.g.: customer.Wallet.RemoveMoney()

Refactoring Types

Demo

Refactoring

Using IDE to do refactoring

Refactoring Patterns

- Large repeating code fragments → extract repeating code in separate method
- Large methods → split them logically
- Large loop body or deep nesting → extract method
- Class or method has weak cohesion → split into several classes / methods

Refactoring Patterns

- 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

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

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

Key Points

- The Code Lifecycles
- Common Code reviews
  - Testing
  - Debugging
  - Static Analytics
- Refactoring as a remedy to improve the code quality by doing static analytics
- Code Smells as indicator to do refactoring
- Refactoring Patterns is ready to use recipes for developer

References

- Kent Beck; Martin Fowler; John Brant; Don Roberts; William Opdyke. Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional, 1999
- Svetlin Nakov;Nikolay Kostov. Refactoring: Improving the Quality of Existing Code. Telerik Academy. 2007.