# Designing with patterns - Refactoring

"Bottom up" based application of patterns

"Improving the design after it has been written"

## What is Refactoring?

- Two definitions, the object and act of change in software
  - A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.
  - To restructure software by applying a series of refactoringns
- A practical definition
  - Systematically alter the source code of a program in order to improve its design
    - Goal of change is to make software easier to understand and modify, not e.g. change it for better performance.
  - All changes are correctness preserving
  - Changes are done one step at a time
  - Frequent regression testing is used to enable changes without too much bug fixing.

## Why to refactor?

- Refactoring is just one of the ‘good software engineering practices’, no magical tool. It should be used for several purposes.
- Refactoring improves the design of software
  - The design of software will decay – as changes are done the code (and design) loses its original structure. The self-documenting property of the code vanishes, the design is not any more readable in the code.
  - Refactoring does the opposite, it creates design in the existing code.
  - Regular refactoring adjusts the design piece by piece to changing functionality

## Why to refactor?

- Refactoring makes software easier to understand
  - The task of getting the software to do its required task is often all the pain you can take, you really do not want to think about the future developers or maintainers pain.
  - You should, code is written once but read many times.
  - Refactoring lets you concentrate on getting the functionality in place, and then afterwards concentrate on easing the life of future developer.
  - The code should communicate your design ideas.

## Why to refactor?

- Refactoring helps you find bugs
  - When refactoring, you deeply understand what the code does, clarify the purpose of code to a point where you simply can not avoid seeing the bugs.
- Refactoring helps you program faster
  - Without refactoring you start faster but loose speed after a while.
  - Good quality, good and up to date design, readability, self-documenting code, rapidly found bugs … all this is necessary to maintain speed in software development.
- All of the above is achieved by removing duplicate code, localizing code. Thus this is a general goal of refactoring.
- The final reason: because you will finish the project earlier.

## When to refactor?

- Simple answer: *all the time*
  - Refactoring should not be a phase that is executed e.g. every two weeks, but an integral part of design/programming.
- More specifically, think refactoring when …
- *Rule of three*
  - first time you do something, just do it. Second time you do something similar, you may do it with duplication. Third time something similar, refactor.
- Refactor when you add functionality
  - Refactor the code that is going to change before you make the change in order to deeply understand the code
  - If the change does not fit in easily, refactor the design to enable smooth addition of the new feature
- Refactor when you need to fix a bug
  - The fact that there was a bug states that the code is not easy to understand, otherwise the bug would not have born
- Refactor in a code review
  - If you are going to go through the code, why not go through the design as well, and to deeply understand the code

## Code smells – when to apply a refactoring

- It is easy to say how to improve a certain local design, but far more difficult to say when the design is poor enough that you should improve it.
- Code smells are symptoms, something that should raise your attention to ponder the need for refactoring.

- Duplicate code – number one smell
  - example: you have similar code in two sibling subclasses. Use *Extract Method* to separate similar parts from different bits, then you can use *Pull Up Field,* or may find *Form Template Method* suitable, or maybe *Substitute Algorithm* is what you need.
  - example: you have duplicate code in unrelated classes. Consider using *Extract Class* in one class and then use the new component in the other. Other possibility is that the code really belongs only to one class and should be invoked by other, or that it should be in a third class and referred to by both original classes.
  - The cure is up to you, you have to analyze the situation and decide where to put the code.

## Code smells – when to apply a refactoring

- Long method
  - *Extract Method, Replace Method with Method Object, Introduce Parameter Object...*
- Large class
  - *Extract Class, Extract Subclass, Extract Interface*
- Divergent change (same parts change for many reasons)
  - This is typically a violation of SRP or a non-cohesive class
  - *Extract Class*
- Shotgun surgery (changes are not local)
  - *Move method, Move Field, Inline Class…*
- Feature envy (a class is too interested on another classes features)
  - *Move Method, Extract Method*
  - *Strategy, Visitor*
  - Fundamental rule: put things together that change together

- … and many, many more…

## Testing

- Refactoring is *impossible* without automatic, comprehensive tests.
- Writing test must be integrated to writing code.
- Before refactoring, write tests to show if you did the job correctly
- Before fixing a bug, write tests.

- JUnit is one suitable testing framework for this purpose.

## Refactoring to patterns

- Patterns define higher-level goals to refactorings
  - A pattern solution to a code smell can not be achieved with just one or two low level refactorings but a sequence of them
- Patterns define direction of refactorings
  - Refactor towards a pattern, not necessary to refactor all the way to the pattern solution
- Patterns help you to understand the forces behind code smells
- Patterns help you see the design that mitigates the forces