# Introduction to Refactoring

Refactoring involves improving the internal structure of code without altering its external behavior.

## Q & A about Refactoring

Q: Is bug fixing considered refactoring?

A: No. Bug fixing changes how the software behaves externally, so it does not qualify as refactoring.

Q: Is adding new features considered refactoring?

A: No. Adding features introduces new behaviors or changes existing ones, which goes beyond the scope of refactoring.

Q: Is source code organization refactoring?

A: Not necessarily. Source code organization can improve the internal structure of code, but it can also introduce alterations to external behavior.

## Refactoring Companion

- Unit Test
- Version Control System

## Unit Test

- To make sure that its external behavior is not altered, the code should pass its unit tests.
- For refactoring, a unit test is a must, not a choice.

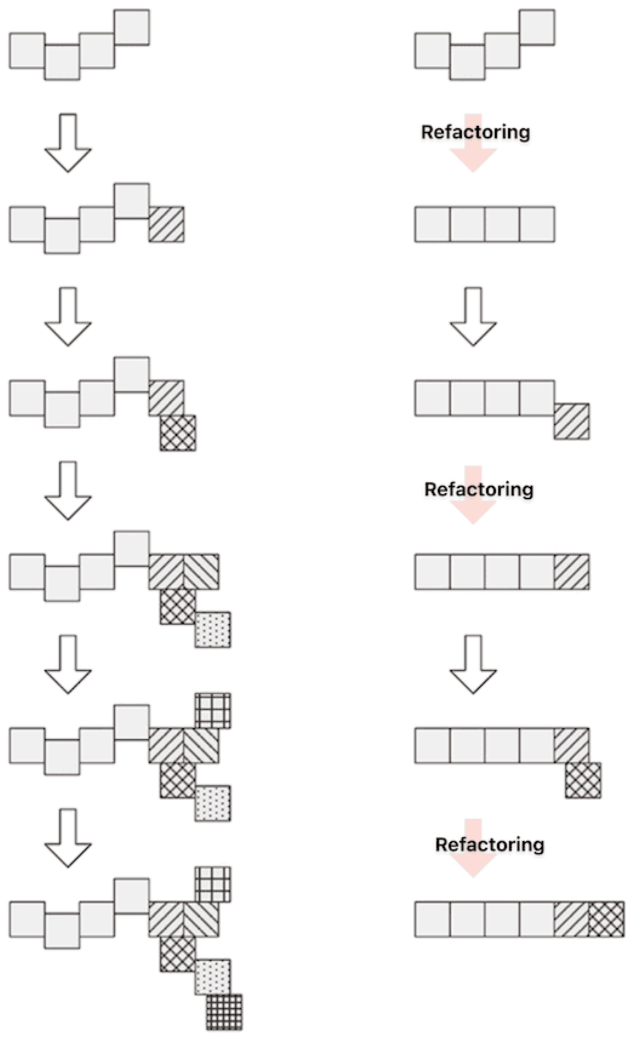
## Version Control System

- We must revert to any commits after refactoring.
- We should commit the changes to VCS, such as Git/GitHub, so we can `checkout` the commit.

## Goal of Refactoring

- Making bug fixing easy
    - Bug fixing is not refactoring, but with refactoring, we can make bug fixing easy.
- Making feature addition easy
    - When we add features, the code becomes more complicated and complex to maintain.
    - We should refactor the code to make it maintainable.

## Source code maintenance without/with refactoring



## Software Design and Refactoring

- Without software design, we cannot effectively refactor code.
- **Software design** is the **guideline** to refactor code.

## Limitations of Refactoring

- The code should be working before starting refactoring.
- It is not wise to refactor code when we should ship the software ASAP.
- It takes time to see the importance of refactoring.

## Code Smell and Refactoring

- Hard to read code
- Hard to change code
- Hard to extend code

## Six code smell keywords

- Duplication
- Too long
- Too many
- Too public
- No matching name
- No OOP like

## Duplication

- Implies multiple impacts from a single change
- Implies copy & paste coding
- Implies no software design

## Duplication: Solution

- Extraction
- Introduce Null Object
    - with multiple null checking
- Error code to exception
    - with multiple error checking

## Too long

- Hard to read
- Hard to understand
- Maybe, a violation of SRP

## Too long: Solution

- Make it small and short
  - Method extraction
  - Class extraction

## Too many

- Method or class extraction may end up ``Too many methods and classes."
- Too many classes in a package

## Too many: Solution

- Remove the middleman
- Class inline
- Method inline

## Too public

- Violation of Encapsulation.
- Mainly because `private` is too inconvenient.
- Any object can access the field or method to cause surprises.

## Too public: Solution

- Information hiding
- Field encapsulation
- Constructor to factory method
- Avoid magic number

## No matching name

- Wrong names can lead to confusion
- The updated code might do something different from the initial name.

## No matching name: Solution

- Rename class
- Rename method
- Introduce name variables
- Separation of temp variables

**No OOP like**

- If/switch statement
- checking types all the time

## No OOP like: Solution

- Code to class
- Selection code to sub-classes

## Refactoring Catalog

- We have many more code smells and refactoring patterns.
- Use the link https://www.refactoring.com/catalog center

## Systematic Refactoring

- *Identify* the refactoring approach
- Refactor code *step by step*
- Refactoring is an *accumulation* of problem-solving experience
  - Use it for solving your problems

## Refactoring rules

*1. Step by Step: make single changes*

```
# Wrong
A1 -> B1 -> A2 -> A4 -> B2 -> B3 -> A4 -> B4

# Correct
A1 -> A2 -> A3 -> A4
B1 -> B2 -> B3 -> B4
```

## *2. Git and Unit Tests*

```
# Ready to go back
A1 (test & commit) -> A2 (test & commit)
-> A3 (test) # something wrong
-> A2 (find some issues)
-> A4 (solve and commit)
```