Clean Code - Tutorial

Introduction

This tutorial provides an overview about the most important Clean Code rules.

In the following, steps of how to refactor bad code are listed and explained.

Important Note:
The tutorial is just a guide of how to refactor very bad code. In reality, one does not have to follow all those steps most often, because probably, code exhibits just some small issues.
Nonetheless, every developer should be aware of all those principles, rules and refactoring steps.

Clean Code Principles

SRP – Single Responsibility Principle

Every data structure and functionality should be responsible for one single purpose only.
In practice, this means that an extensive issue needs to be split into small tasks, which can be dealt with in clearly structured programs.
Most often, one applies this principle to classes, methods and data structures so that every single component serves only one specific purpose. If a component serves more than one purpose (e.g. calculate numbers and save them in a file), one has to split the component into smaller sub-components.

DIP – Dependency Inversion Principle

Complex programs usually have their own data structure so that modules of a high level are not mixed with modules of a low level. The principle expresses that modules of different levels always should be separated. As an effect when applying this principle to code, one will be able to see a clearly structured code which can be read like a newspaper:
The head-module contains general information. The more one follows the abstraction-structure, the more details will become visible.

OCP – Open Closed Principle

The Open Closed Principle deals with inheritance and the idea of making software open for extensions but closed for modifications as well. The aim of this principle is to keep the core as a basic component and add features without touching the core. In OOP, inheritance is the means of choice. Using inheritance, one is able to define basic program components which can be expanded afterwards.

Refactoring – Tutorial

Refactoring will always be an important part of developer's work in order to keep code clean. Refactoring consists out of more than four important aspects, but in the following, we will focus on meaningful names, methods, comments as well as formatting and how to match their requirements.

Meaningful Names

- Choose **describing variable-/class-names** (as you would comment this variable/class)
- Choose **pronounceable** names (e.g. "receiveUsernameFromLogin" instead of "rcvUsrFLog")
- Avoid names which contain "Controller", "Manager",… → they express more than 1 purpose!
  Avoid names which contain datatype-names such as "List", "Array", "String",…
- Avoid single-digit names!
- Is your code fluently readable like a newspaper? If not: Refactor again!

Concept of Methods

- Size of Methods: Max. 50-100 lines
- Only **one task** per method!
- Name of method: **Verb** ("get", "set", "add", "delete", …) + **purpose**
- Do not mix high-abstraction-levels with low-level-modules
- Use as **few parameters** as possible

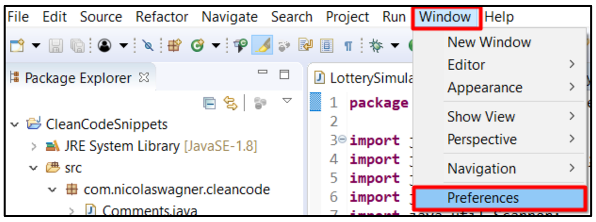
Comments

- Do not write code-describing comments (the code itself should do this for you!)
- Appropriate comments:
  - Law (e.g. license)
  - Informative (e.g. why an unusual way of implementation has been chosen)
  - Warnings (e.g. warning of long run-time)
  - TODOs
- JavaDocs should be maintained (for public APIs)
- Never comment out some code!

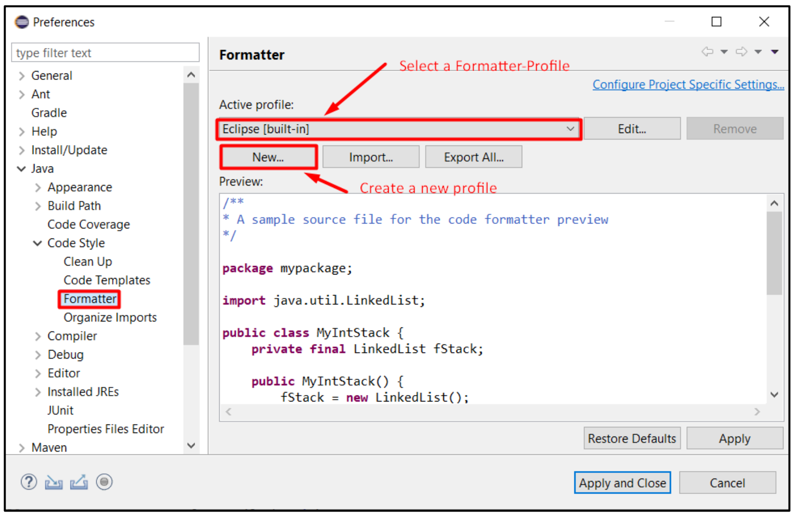
Formatting

- File-size: maximum of 200-500 lines
- File-structure:
    - At the top: General code
    - At the bottom: Most detailed code
- Keep up one single code style consistently (concerning line breaks, whitespaces, indents, …)
- Use of Formatter (here: Eclipse):

1. Open a workspace using Eclipse
2. Click on “Window” → “Preferences”



3. Go to “Java” → “Code Style” → “Formatter”



```
/**
 * A sample source file for the code formatter preview
 */

package mypackage;

import java.util.LinkedList;

public class MyIntStack {
    private final LinkedList fStack;

    public MyIntStack() {
        fStack = new LinkedList();
```

4. Select a given profile or create your own one
5. Format your code using CTRL+SHIFT+F

When working in a team, it makes sense to share a formatter with every team member. The procedure of activating and editing a Formatter should be analogical in other common development environments.

Tests

As JUnit-Tests are an indispensable part of software development nowadays, make sure to write tests for extensive projects.
As there is another e-Portfolio for JUnit-Testing, check it out as well:
http://softwareengineering.freeforums.net/thread/729/unit-testing-junit