**Code Refactoring**

**Overview**

1. **What is code refactoring?**
2. **Importance of Refactoring**
3. **When to refactor code**
4. **Techniques used when doing code refactoring**
5. **Code refactoring Workflow**
6. **Performing Code Refactoring(how I did it)**

**Reference link:** [**https://martinfowler.com/articles/workflowsOfRefactoring/#2hats**](https://martinfowler.com/articles/workflowsOfRefactoring/%232hats)

**What is code refactoring?**

The process of restructuring computer code without changing or adding to its external behavior and functionality.

The changes in existing source code preserve the software’s behavior and functionality because the changes are so tiny that they are unlikely to create or introduce any new errors.

**The importance of code refactoring**

 The main purpose of code refactoring is to turn dirty code into clean code, which reduces a project’s overall technical debt.

Dirty code is an informal term that refers to any code that is hard to maintain and update, and even more difficult to understand and translate.

Dirty code is typically the result of deadlines that occur during development – the need to add or update functionality as required, even if its backend appearance isn’t all that it could or should be.

This is the idea behind technical debt: if code is as clean as possible, it is much easier to change and improve in later iterations so that your future self and other future programmers who work with the code can appreciate its organization

Dirty Code can include the following;

* Codes, methods, or classes that are so enlarged that they are too unwieldy to manipulate easily
* The incomplete or incorrect application of object-oriented programming principles
* Superfluous coupling
* Areas in code that require repeated code changes in multiple areas in order for the desired changes to work appropriately
* Any code that is unnecessary and removing it won’t be detrimental to the overall functionality

Clean code, on the other hand, is much easier to read, understand, and maintain, thereby easing future software development and increasing the likelihood of a quality product in shorter time.

**When to refactor**

 If you’re the developer, you already know where you may have cut corners in your code in order to create the functionality you needed.

It is very easy for the developer where he or she might have made cuts in the code to create a functionality so it becomes very easy for them to know when and where or how to refactor that code. But if it is just a member of the team or a new developer on the project, it is always harder to prioritize code refactoring.

So when do we refactor, the following are some of these instances;-

* **Refactor in accordance with the Rule of 3:**
  + The first time the developer is coding, they should just get it done, even if it’s with dirty code, so the software functions as needed.
  + The second time they are doing a similar change, they can do it again the same way they will know it a little better, so they may be speedier but the code still won’t be perfectly clean.
  + When they encounter this change for the third time, refactoring can now start.
* **Refactor during code review** – the last chance to clean up code before it is live. Developers always try doing a two-person review so they can fix quick, low-hanging fruit and then better gauge which difficult code change areas are worth the time.
* **Refactor during regularly-scheduled intervals.** This doesn’t have to mean dedicating a whole day to it, but rather developers add it as part of their routine – spending the last hour of a workday on refactoring.

**Techniques used when doing code refactoring**

The techniques used for code refactoring do vary depending on the problems in your code. Here are common techniques.

* **Correcting the composing methods** in order to streamline, removing duplication, and make future changes a lot easier in your code.
* **Simplifying conditional expressions**, which become unnecessary complex over time, and method calls so they are easier to understand, improving interfaces for class interaction.
* **Moving features between objects** in order to better distribute functionality among classes. This can include safely moving functionality, creating new classes, and hiding implementation details.
* **Organizing data** to improve handling and class associations so that the classes are recyclable and portable.

**Refactoring checklist**

This helps to determine whether the code is clean enough.

* Is the code obvious to other developers?
* Are there no more duplications in the code?
* Does the code contain minimal classes?
* Does it pass all the tests?
* Is the code easier to maintain?

**Code refactoring Workflow**

Code refactoring has the following workflows as envisaged by Martin Fowler

1. TDD Refactoring
2. Litter-pickup refactoring
3. Comprehension Refactoring
4. Preparatory Refactoring
5. Planned Refactoring
6. Long term Refactoring

**TDD Refactoring:**

When refactoring every change you make is a small behavior-preserving change. You only refactor with green tests, and any test failing indicates a mistake. By stringing together a series of small changes like this you can move more quickly and with less risk because you shouldn't get trapped in debugging.

**Green**

➋ **Make it work:** Make the failing test pass (go green) by implementing the necessary functionality simply but crudely.

➊ **Add a Test:** adding a test for yet-to-be-built functionality. This test will fail, making the test suite red.

➌ **Make it Clean:** Use refactoring to ensure the overall code base is as clean and well-designed as possible for currently-implemented functionality.

Test-Driven Development (TDD) is often described in terms of the red-green-refactor cycle

**Refactoring**

**RED**

**Performing Code refactoring**

I performed code refactoring by restructuring the following programs.

1. A python game.
2. Web scraper, a public github repository
3. Library Management system, a public gitlab project.