ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN

EXERCISE 04

REFACTORING

SOFTWARE ANALYSIS AND DESIGN

21120280 – Lý Minh Khuê

Thành phố Hồ Chí Minh – 2024

TABLE OF CONTENTS

CONCEPT OF REFACTORING

Refactoring is a technique that turn your messy codebase into a nice piece of "clean code", without changing any of its external behavior.

Refactoring does not change any of the functionality of the system, only concentrating in altering its internal structure.
Each refactoring contributes a little to the whole process of refactoring the codebase. A series of these refactorings should be the thing we actually derive for, it produces a significant structuring. Since each refactoring is rather small, but in fact it's less prone to fail the function, the system remains operational after each refactoring is applied.

TECHNICAL DEBT

1. What is technical debt:

Why does **Refactoring** matter? Or why should we refactor our code?
A term was introduced to explain the reason behind this: "**Technical debt**"

"Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite… The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation, object-oriented or otherwise."

Ward Cunningham

This debt implies the expense you have to charge later when choosing the easy path instead of another that is optimized but take more time. So what you have to repay later is the reworking on the code for maintenance and scalability purposes.

Equivalent to monetary debt, when you get a loan from the bank, you have to pay the additional interest afterwards monthly, if you don't the interest would be accumalted. Unpaid technical debt increases software entropy(deterioration of software quality), and the future work cost would be disturbing.

For example: when we have a bug during our coding time, we use try/catch to swallow the exception(hotfix-ing) rather than actual fixing or refactoring it.

2. What are the causes:

- Business pressure: Sometimes business circumstances might force you to get features to be released sooner than expected. This is what pushes devs to prioritize speed over code quality.
- Test suite shortage: Lack of a test suite, this will encourage hotfix or quick patches

- Delayed refactoring: When the project evolves, certain part of the code become outdated, on the queue to be redesigned to match the new requirements. However, other programmers who works on a different part depend on these obsolete parts, which makes the refactoring process delayed, the more delayed the more debt the team has to repay later on.
- Incompetence: The working team doesn’t have the ability to write decent clean code after all.
- Poor leadership: The leader has no knowledge about technical debt, gives unthorough commands to the team
- Lack of collaboration: where knowledge isn't shared around the organization and business efficiency suffers, or junior developers are not properly mentored.
- Lack of software documentation, where code is created without supporting documentation. The work to create documentation represents debt.

3. Consequences:

- Reduced speed: Technical debt can slow down development velocity and diminish the flexibility to respond quickly to changing business requirements.
- Increased maintenance costs: When technical debt accumulates, the cost of maintaining the software typically increases over time. Devs would spend more time fixing bugs, and addressing issues.
- Accumulated technical debt increases bugs, regressions, and unexpected behavior into the software.

WHEN TO REFACTOR

1. Rule of Three(Three strikes and you have to refactor):

It is said to be two instances that use the same piece of codes, do not need refactoring, although it is hard to do it anyways. However, when you are doing that same thing the third time, you definitely need to refactor, extract that code into function or module. This would tell you should not violate the **DRY principle (Don't Repeat Yourself).**

2. When fixing a bug:

In order to address and fix a bug, the first thing you should do is refactoring. By that, it increases the chance of the bug revealing itself.

3. Before passing the code to someone else or before reviewing:

This could make the code legible to the receivers. If there is any changes required, you would also easy to spot it and fix it quickly.

REFACTORING STRATEGIES

1. Breaking code into smaller pieces:

Applying DRY principles, breaking down code to reusable units that are clear, well-defined, is one of the best practices to refactor.

Extracting modules moves part of code of the existing class to a new class or interface.

Extract one big function into smaller ones to make it legible.

2. Improving names and location of code:

- Move methods and fields into a more appropriate modules and directories.
- Rename method or rename field – changing the name into a new one that better reveals its purpose

3. New functionality should not be created:

During refactoring, even if you see a glimpse of able to apply another functionality into that piece of code, just do not do it. Try to concentrate on the refactoring process only, and writing these functions afterwards for another commit.

4. The passed tests should be passed again:

You must not make errors out of refactoring, or else what was the point of it. Go ahead and fix the error before transform other code.

All the coverage test should be covered.

REFERENCES

https://refactoring.guru/refactoring/what-is-refactoring
https://viblo.asia/p/technical-debt-no-ki-thuat-no-code-khong-chi-tra-bang-code-nwmGyEQMGoW
https://refactoring.com/
https://stringee.com/vi/blog/post/refactoring-code-la-gi