## **Refactoring: Improving the Design of Existing Code**

## **Chapter 1: Refactoring, a First Example**

With the first example, there is sample program about Movie Rental. The first sample program is quite dirty and not quite understandable. The sample code is not well-written by the programmer. The program may not be broken, but it does hurt. It is making your life more difficult because you find it hard to make the changes your users want. This is where refactoring comes in. When additional feature is needed, refactor the program to make it easy to add a new feature, then add the new feature. Before refactoring, it's important to have a solid set of tests in place to ensure that the changes don't cause any issues. With long method, it should be decomposed and redistribute. Smaller pieces of code tend to make things more manageable. They are easier to work with, and move round.

There are types of methods that have been used on this first example: the **Extract Method.** This extracts a method, where you will know what can go wrong. This way, you will know how to refactor safely without (avoid) any problems or error; **Move Method**. If the method is on the wrong object, you can use this method to move the class to the class which is should belongs to. This might be a time-consuming process, then it benefits you to have an organize and better code, as well reduce repetition of code; and **Replace Conditional with Polymorphism**. Instead making a switch statement, make a class for each condition. The result is that the proper implementation will be attained via polymorphism depending on the class. When there is a new condition, all you need to do is add a new subclass without touching the existing program code.

These methods help distribute responsibilities more effectively, make the code more concise, clear, and readable, and simplify the process of maintaining the program.

When making changes, it's essential to follow a pattern of testing, making small changes, and testing again. This approach helps you move quickly and safely in case of any issues.

## **Chapter 2: Principles in Refactoring**

There are two definitions of Refactoring depends on what context.

1. **Refactoring (noun):** a change made to the internal structure of software to make it easier to understand a cheaper to modify without changing its observable behavior.
2. **Refactor (verb):** to restructure software by applying a series of refactoring without changing its observable behavior.

Refactoring is not just about cleaning up code, it is a technique for cleaning up code in a more controlled and efficient manner. The goal of refactoring is to make software easier to understand and modify.

Like refactoring, performance optimization also alters the internal structure of a component, but unlike refactoring, it often makes code harder to understand. Despite this, it is still necessary for achieving the desired performance. It is important to remember that refactoring does not change the observable behavior of software. The software still performs the same function as before.

The "two hats" metaphor by Kent Beck suggests that when developing software, time should be divided between two activities: **adding function** (new features), and **refactoring** (restructuring code). Only tests should be changed if necessary to accommodate changes in interfaces. Once the code is better structured, a new function can be added.

Refactoring is a valuable tool that can, and should, be used for several purposes. It’s a pair of silver pliers that helps you keep a good grip on your code. **Why you should refactor?**

1. Without refactoring, the design of the program will decay. **Regular refactoring helps code retain its shape**, and **improves the design of software**.
2. It does matter if it takes a programmer a week to make a change that would have taken only an hour if you had understood your code. A little time spent refactoring can make the code better communicate its purpose, it **makes the software easier to understand**.
3. Understanding the code also **helps you spot bugs**. Once you refactor the code, you can deeply understand what the code does.
4. It helps you **develop code more quickly**. Good design code is essential for a rapid software development and stops the design of the system from decaying.

**When is the time to refactor your code?**

1. Three strikes and you refactor.
2. Refactor the code before adding a new feature to make the process easier and more efficient.
3. Fixing a Bug.
4. As you do a Code Review.

Our job is to build effective software as rapidly as we can. Software developers are professionals. A schedule-drive manager wants you to do things the fastest way you can, and the fastest way is to refactor.

Refactoring can be challenging, particularly when changing a database schema or published interface. Some design changes may also be difficult to refactor. Refactoring should not be done when close to a deadline, as there may not be enough time to refactor effectively.