6 Code Refactoring Techniques You Should Know

1. Red-Green-Refactor

This technique is an integral part of test-driven development (TDD), a software development methodology that emphasizes writing tests before writing the actual code. The process consists of three main steps:

- Red: Write a failing test that exposes the issue or required functionality. The goal is to create a test that will fail because the desired functionality has not been implemented yet. This step ensures that the test is valid and can detect the absence of the intended feature.
- Green: Write the minimum code necessary to pass the test. In this step, you focus on implementing the functionality needed to make the test pass, without worrying about the quality or maintainability of the code. The primary goal is to make the test pass as quickly as possible.
- Refactor: Improve the code while keeping the test green (i.e., without breaking the functionality). Once the test passes, you can refactor the code to make it cleaner, more efficient, and maintainable. The test serves as a safety net during the refactoring process, ensuring that the external behavior remains unchanged.

This cycle of red-green-refactor is repeated for each new feature or bug fix, promoting a development process where the code is continuously tested and refactored, resulting in higher-quality and more reliable software.

2. Refactoring by Abstraction

Refactoring by abstraction is a technique where you identify common functionality shared by multiple classes or methods and extract it into a separate, abstract class or

interface. This process helps reduce code duplication, promotes reusability, and makes it easier to manage and maintain the shared functionality.

For example, consider two classes with similar methods that perform the same calculations. By extracting the shared logic into an abstract class or an interface, you can create a single implementation that both classes can inherit or implement. This abstraction reduces the amount of duplicated code and makes it easier to update or modify the shared functionality in the future.

3. Composing

This refactoring technique focuses on breaking down large classes or methods into smaller, more manageable components. The primary goal is to improve code readability, maintainability, and testability by making sure each component has a single responsibility.

This approach promotes a modular and organized codebase, making it easier to understand, modify, and extend in the future. Composing encourages a clean separation of concerns, which results in higher-quality and more reliable software.

4. Simplifying Methods

This refactoring technique aims to make methods more understandable and maintainable by reducing their complexity. It involves various approaches that simplify the code, such as reducing the number of method parameters, replacing complex conditional logic with simpler constructs, or breaking down long methods into smaller, focused methods.

5. Moving Features Between Objects

This technique focuses on reassigning responsibilities or methods between classes to improve cohesion (how closely a class's responsibilities are related) and reduce coupling

(the degree to which one class depends on another). By redistributing code or functionality among classes, you create a more balanced and logical design that is easier to maintain and extend.

6. Preparatory Refactoring

This approach is used when preparing the codebase for new features or significant changes. The objective is to create a cleaner, more adaptable foundation for future development by restructuring code, updating deprecated APIs or libraries, or simplifying complex logic. Preparatory refactoring helps ensure the codebase remains maintainable, testable, and scalable as new functionality is added.