Refactoring Cheat Sheet

A concise cheat sheet covering essential refactoring techniques, principles, and tools for improving code quality and maintainability.

Core Principles

Definition

Refactoring: Improving the internal structure of existing code without changing its external behavior.

Benefits

Improved Design	Easier to understand, modify, and extend the code.
Reduced Complexity	Simplifies code, making it less prone to errors.
Enhanced Maintainability	Reduces the cost of future development and bug fixes.
Increased Performance	Can sometimes improve code execution speed.

When to Refactor

- **The Rule of Three:** Refactor after you've done something similar three times.
- **When Adding Functionality:** Refactor to make it easier to add new features.
- **When Fixing a Bug:** Refactor to prevent similar bugs in the future.
- **During Code Review:** Identify areas that can be improved.

Key Refactoring Techniques

Composing Methods

Extract Method	Create a new method from a code fragment. **Example:** Isolating a complex calculation into its own function.
Inline Method	Replace a method call with the method's content. **Example:** Removing a simple method that doesn't add value.
Replace Temp with Query	Replace a temporary variable with a method. **Example:** Calculating a value on demand instead of storing it.

Moving Features Between Objects

Move Method	Move a method to another class that it uses more. **Example:** Moving a method that uses more features of another class to that class.
Move Field	Move a field to another class that it is used by. **Example:** Moving a field to the class where it's primarily accessed.
Extract Class	Create a new class and move related fields and methods from an existing class. **Example:** Separating UI logic from business logic.
Inline Class	Move all features from a class into another. **Example:** When a class is no longer complex enough to warrant its own existence.

Organizing Data

Replace Data Value with Object	Replace a data value with an object. **Example:** Using an object to represent a simple value like a phone number or zip code.
Change Value to Reference	Change a value object to a reference object. **Example:** Using a single `Customer` object instead of creating new ones with the same data.
Change Unidirectional Association to Bidirectional	Add a back pointer in association. **Example:** Making parent and child objects aware of each other.

Simplifying Conditional Expressions

Decompose Conditional

Description	Separate the 'then' and 'else' parts of a conditional into distinct methods.
Motivation	Improves readability and allows for easier modification of individual branches.
Example	Turning a large if-else block into smaller, named methods.

Consolidate Conditional Expression

Description	Replace a sequence of conditional expressions with a single conditional expression.
Motivation	Makes the code easier to understand when multiple conditions lead to the same result.
Example	Combining several `if` statements that return the same value.

Replace Nested Conditional with Guard Clauses

Description	Replace nested conditional statements with guard clauses.
Motivation	Makes the code more readable by exiting early for special cases.
Example	Using `return` statements at the beginning of a method to handle edge cases.

Dealing with Inheritance

Pull Up Field

Description	Move a field to the superclass.
Motivation	Eliminates duplication when subclasses have the same field.
Example	Moving a common property like `name` to the parent class.

Pull Up Method

Description	Move a method to the superclass.
Motivation	Avoids code duplication when subclasses have similar methods.
Example	Moving a `calculateSalary` method to the parent class.

Push Down Method

Description	Move a method from the superclass to subclasses.
Motivation	Allows specialized behavior in subclasses without cluttering the superclass.
Example	Moving a specialized method like `displayImage` to subclasses that need it.

Replace Inheritance with Delegation

Description	Create a field on the class that refers to the original class and delegate methods to it.
Motivation	Reduces tight coupling between classes and allows more flexible composition.
Example	Instead of inheriting behavior, use an object of another class to perform certain actions.