Clean Code Summary DRAFT

Kevin Löffler

April 2023

Contents

1 About the book

1.1 Introduction

Often ranked as the most important book to read for software engineers "Clean Code" by Robert C. Martin covers fundamental software engineering practices. Martin explains how to write clean code that is, readable, easy to modify, and maintainable by providing practical tips and examples that developers can apply to their projects. The book is organized into chapters that cover different aspects of clean coding, from naming conventions to code structure, and also includes case studies and real-world examples.

In this summary I attempt to capture some of the most important concepts, often with practical examples, because its often easiest to let the code speak for itself. An idea that I believe Robert C. Martin would agree with.

1.2 Clean Code

First we need to agree that good code is important. Bad code can be detrimental to a project, product and ultimately a business itself. Martin tells stories about companies that collapsed because their code base became unmanageable. He identifies several problems with code that make working with it unfeasible and unenjoyable.

1.2.1 Readability

Developers spend a much bigger portion of their time reading code than writing (10:1). CITATION?? Code that is hard to read therefore slows the creation of new features down. The more code that is hard to read the less progress is made overall.

1.2.2 Maintainability

In the beginning of a project teams often progress extremely quickly. Later theirs speed often gets destroyed by unmaintainable code. It manifests itself when a change at one point in the program leads to refactors in several completely different parts of the code. Resulting in the fact that no change is trivial and a developer needs to understand every part of a system to be able to work on it.

1.2.3 Care

Martin says that the core of the problem is not the reasons we often hear: to little time, a budget that is too small, shifting requirements. He proclaims the problem to be more fundamental: unprofessionalism or a lack of care from the developers.

Martin calls Code that satisfies these principles 'clean code'.

2 Naming

2.1 Descriptive names

Names should always describe why they exist and what they do. If a variable declaration needs a comment it has already failed.

Bad and good vriable names

```
let d: int;  // elapsed time in days
let elapsedTimeInDays: int;  // descriptive = no comment needed
```

Additionally, variables should always have names that are pronounceable and searchable. A variable named '`x`' is very hard to ever find again.

2.2 Honest names

When using programming concepts in our names, developers should be careful not to misuse a name. For example a variable that is called '`accountList`' should really be a list and not some other data structure. Much better would be an implementation agnostic name like '`accounts`'.

2.3 Stay in the domain

When naming variables and functions, a developer should stay in the current problem domain. For example when building a photo editing app we should use terminology that professionals in that area use. The clone tool should be implemented with a clone class, not the copy or replace class and the variable should be called opacity not transparency.

2.4 Classes and methods

Class names should be nouns like `Customer`, `AboutPage` or `FileParser`. Methods should be verbs like `fetchRequest`, `deletePage` or `save`.

2.5 One concept per word

Each abstract concept should have one word describing it. Methods that do something similar should use the same word. For example there should only be one word to describe getting data from somewhere: `fetch`, `get` or `retrieve`.

2.6 Add context if necessary

Sometimes the context of a variable is clear: `firstName`, `lastName`, `street`, `city`, `state` form an address. But if there is a method that only uses `state` it would not be immediately clear what it means. The best solution would be to group all of them in an address class but if that is not possible, a prefix could be usefull: `addrState`.

3 Methods

3.1 Short

Martins most important rule is that methods should be short, ideally under 20 lines.

3.2 One Task

Each method should do one thing but do it well. Having methods that are short and specific in combination with descriptive names leads to self documenting code. Doing one thing also means that there should not be any deep nesting. Especially bad are different layers of abstraction in the same method.

```swift
func importImage(imagePath: String) {
    var image: Image? = nil

    if imagePath.startsWith('.') -> Image? {
        // relative path
        try {
            image = readImage(from: "/usr/lib/photos/\(imagePath)")
        } catch err {
            throw InvalidPathException()
        }
    } else {
        // absolute path
        try {
            image = readImage(from: imagePath)
        } catch err {
            throw InvalidPathException()
        }
    }

    if image.colorSpace != "RGB" {
        image?.transformColorSpace(to: "RGB")
    }

    return image

}
```

Listing 1: Method with more than one task

```swift
func importImage(imagePath: String) -> Image? {

    guard validateImagePath(imagePath) else {
        throw InvalidPathException()
    }

    let rawImage: Image = readImage(from: imagePath)
    let image: Image = normalizeColorSpace(image: rawImage, to: "
    RGB")

    return image
```
Listing 2: Refactored method that only does one thing: import an image

The refactored method improves the code in different ways:
1. By treating error handling as a 'task' we need to separate it out into another method. The new method is much more readable without the nested conditionals and error handling.
2. With the Introduction of the `normalizeColorSpace` method, the code becomes more reusable because we can use this method with a different color profile somewhere else in the code.