T 04 Modul 450: CleanCode

Guter Code zeichnet sich nicht nur durch Funktionalität aus, sondern auch durch Lesbarkeit, Wartbarkeit und Verständlichkeit.

Inhaltsverzeichnis

1. Die Bedeutung von Code

Code ist die Sprache, mit der Entwickler Probleme lösen und Maschinen anweisen, Aufgaben auszuführen. Doch nicht jeder Code ist gleichwertig, die Qualität des Codes hat einen direkten Einfluss auf:

- die Produktivität von Teams
- die Fehleranfälligkeit von Anwendungen und
- die langfristigen Wartungskosten eines Projekts

1.1. Die Entstehung von Code

Das Schreiben von Code ist ein kreativer Prozess, der sowohl technisches Wissen als auch ein Gespür für Struktur und Klarheit erfordert. Doch was zeichnet guten Code aus? Hier helfen uns Zitate führender Köpfe der Softwareentwicklung, um ein Bewusstsein für den Wert von qualitativ hochwertigem Code zu schaffen.

1.1.1. Zitate zur Inspiration

Martin Fowler: Verständlichkeit vor allem

> "Jeder Trottel kann Code schreiben, den ein Computer versteht. Gute Entwickler schreiben Code, der von Menschen verstanden werden kann."

– Martin Fowler

Interpretation: Der Fokus liegt auf Lesbarkeit und Verständlichkeit. Code wird nicht nur von Maschinen ausgeführt, sondern auch von Entwicklern gelesen, erweitert und gewartet.

Dan Salomon: Die Kosten schlechter Entscheidungen

> "Sometimes it pays to stay in bed on Monday, rather than spending the rest of the week debugging Monday’s code."

– Dan Salomon

Interpretation: Ungenaue oder hastig getroffene Entscheidungen beim Programmieren führen zu Fehlern, die mehr Zeit und Ressourcen kosten, als das Problem von Anfang an richtig anzugehen.

Anonymous: Die Konsequenzen von Fehlern

> "Programming is similar to sex. If you make a mistake, you have to support it for the rest of your life."

– Anonymous

Interpretation: Fehler im Code können weitreichende Folgen haben und langfristig schwer zu beheben sein. Qualitätssicherung und präzises Arbeiten sind entscheidend.

2. Codequalität

2.1. Indikator: Die "WTFs pro Minute"-Regel

Ein beliebter (und humorvoller) Indikator für schlechte Codequalität ist die Anzahl der "WTFs" pro Minute, die ein Entwickler beim Lesen des Codes äussert.

2.2. Die Realität: Lesen vs. Schreiben

Untersuchungen zeigen, dass Entwickler etwa **90%** ihrer Zeit damit verbringen, bestehenden Code zu lesen, und nur **10%** damit, neuen Code zu schreiben.

Konsequenz: Je verständlicher und klarer der bestehende Code ist, desto effizienter können Teams arbeiten.

3. Massnahmen zur Verbesserung der Codequalität

Die Verbesserung der Codequalität erfordert gezielte Massnahmen, die sowohl technische als auch soziale Aspekte berücksichtigen.

3.1. Clean Code

Clean Code ist weit verbreitet in der Softwareentwicklung. Die Prinzipien sind sprachunabhängig und aus diesem Grund interessant für die gesamte Softwareentwicklung.

- Prinzipien:
  - Schreibe einfachen und klaren Code
  - Vermeide unnötige Komplexität
  - Benenne Variablen, Funktionen und Klassen aussagekräftig
- Vorteile:
  - Erhöhte Lesbarkeit
  - Weniger Fehler
  - Einfachere Wartung

3.2. Refactoring

Mittels Refactorings wird bestehender Code in eine neue Form gebracht.
Grundlegend sollte brücksichtigt werden, dass Refactorings als 'sicher' eingestuft werden können,

wenn diese durch die Entwicklungsumgebung durchgeführt werden können.

- Definition: Die Struktur des Codes wird verbessert, ohne dessen Funktionalität zu verändern
- Ziel:
  - Reduzierung von Redundanzen
  - Verbesserung der Architektur
  - Bessere Lesbarkeit und Wartbarkeit
- Beispiele:
  - Aufteilen von langen Methoden
  - Konsolidierung von Duplikaten

Folgende Varianten von Refactorings können angewendet werden:

3.2.1. Extract Method

Teilt komplexe Methoden in kleinere, spezialisierte Funktionen auf.
Dies verbessert die Lesbarkeit und Wiederverwendbarkeit des Codes.

3.2.2. Inline Method

Fügt eine überflüssige oder zu einfache Methode direkt in den Code ein.
Dies vermeidet unnötige Abstraktionsebenen.

3.2.3. Rename Method / Variable

Ändert den Namen einer Methode / Variable, um ihre Funktion besser zu beschreiben.
Dies erhöht die Verständlichkeit des Codes

3.2.4. Entfernung temporärer Variablen

Ersetzt temporäre Variablen durch direkte Ausdrücke oder Anfragemethoden.
Reduziert die Komplexität und verbessert die Übersichtlichkeit

3.2.5. Strukturelle Techniken

Zerlegung von Klassen

Teilt grosse Klassen in kleinere, verständlichere Einheiten auf.
Dies Verbessert die Modularität und Wartbarkeit des Codes.

Branching-by-Abstraction

Ermöglicht schrittweise Änderungen in grossen Systemen.
Dies ist nützlich bei Änderungen, die Klassenhierarchien und Vererbung betreffen.

3.2.6. Bedingungs-basierte Techniken

Vereinfachung bedingter Ausdrücke

Zerlegt komplexe Bedingungen in einfachere Teile.
Dies verbessert die Lesbarkeit und Wartbarkeit von Kontrollstrukturen.

Ersetzen von Fallunterscheidungen durch Polymorphie

Verwendet objektorientierte Konzepte, um komplexe Verzweigungen zu vereinfachen.
Dies erhöht die Flexibilität und Erweiterbarkeit des Codes.

3.3. Pair Programming

- Konzept: Zwei Entwickler arbeiten gemeinsam an demselben Code:
  - Einer schreibt den Code (Driver)
  - Der andere überprüft (Navigator)
- Vorteile:
  - Sofortiges Feedback
  - Geringere Fehlerquote
  - Wissensaustausch

Beim Pair Programming gibt es diverse Varianten wie dies umgesetzt werden kann.

3.3.1. Driver/Navigator-Ansatz

Ein Programmierer (Driver) schreibt den Code, der andere (Navigator) überprüft den Code und gibt strategische Anweisungen.
Die Rollen werden regelmässig getauscht.
Gut geeignet für Anfänger-Experten-Paare

3.3.2. Unstrukturierter Ansatz

Zwei Programmierer arbeiten ad hoc zusammen.
Beide Programmierer sollten ähnliche Fähigkeiten haben Schwieriger für längere Projekte oder Remote-Arbeit.

3.3.3. Ping-Pong-Methode

Ein Entwickler schreibt einen Test, der andere implementiert den Code, um den Test zu bestehen Rollen wechseln sich ab.
Oft in Verbindung mit testgetriebener Entwicklung verwendet

3.3.4. Supported Soloing

Zwei Entwickler arbeiten nebeneinander an unterschiedlichen Aufgaben.

Regelmässige gegenseitige Unterstützung und Code-Reviews Örtliche Nähe ist wichtig für schnelle Hilfe.

3.3.5. Divide and Conquer

Eine Aufgabe wird in kleinere Teile aufgeteilt Jeder Entwickler arbeitet unabhängig an einem Teil Regelmässige Abstimmung und gemeinsames Review am Ende.

3.3.6. Mob Programming

Das gesamte Team arbeitet gleichzeitig an einer Aufgabe. Ein Entwickler programmiert, die anderen navigieren. Die aktive Rolle rotiert zwischen allen Teammitgliedern.
Ziel: kontinuierliche Zusammenarbeit und Abbau von Kommunikationsbarrieren.

Diese Varianten können je nach Teamzusammensetzung, Projektanforderungen und individuellen Präferenzen eingesetzt werden.

3.4. Code Reviews

- Ziel: Code wird von anderen Teammitgliedern überprüft
- Nutzen:
  - Erkennung von Fehlern
  - Austausch von Best Practices
  - Steigerung der Qualität durch mehrere Perspektiven