Clean Code Cheat Sheet

SEITENBAU

Roter Grad

Prinzipien

Don’t Repeat Yourself (DRY)

Jede Doppelung von Code oder auch nur Handgriffen leistet Inkonsistenzen und Fehlern Vorschub.

Keep it simple, stupid (KISS)

Wer mehr tut als das Einfachste, lässt den Kunden warten und macht die Lösung unnötig kompliziert.

Vorsicht vor Optimierungen

Optimierungen kosten immer viel Aufwand. Wer Vorsicht walten lässt, spart oft wertvolle Ressourcen für das, was dem Kunden wirklich nützt.

Composition over Inheritance

Komposition fördert die lose Kopplung und die Testbarkeit eines Systems und ist oft flexibler.

Integration Operation Segregation Principle (IOSP)

Ein deutliches Symptom schlecht wandelbaren Codes sind tiefe Hierarchien funktionaler Abhängigkeit. Sie reduzieren die Verständlichkeit und erschweren automatisierte Tests wie Refactoring.

Praktiken

Die Pfadfinderregel beachten

Jede Beschäftigung mit einem Gegenstand macht ihn zumindest ein klein wenig besser. Ganz ohne bürokratische Planung.

Root Cause Analysis

Symptome behandeln bringt vielleicht schnell eine Linderung langfristig kostet es aber mehr Aufwand. Wer stattdessen unter die Oberfläche von Problemen schaut, arbeitet am Ende effizenter.

Ein Versionskontrollsystem einsetzen

Angst vor Beschädigung eines running system lähmt die Softwareentwicklung. Mit einer Versionsverwaltung ist solche Angst unbegründet.

Einfache Refaktorisierungsmuster anwenden

Code verbessern ist leichter, wenn man typische Verbesserungshandgriffe kennt. Ihre Anwendungsszenarien machen sensibel für Schwachpunkte im eigenen Code. Als anerkannte Muster stärken sie den Mut, sie anzuwenden.

Täglich reflektieren

Keine Verbesserung, kein Fortschritt, kein Lernen ohne Reflexion. Aber nur, wenn Reflexion auch eingeplant wird, findet sie unter dem Druck des Tagesgeschäftes auch statt.

Oranger Grad

Prinzipien

Single Level of Abstraction (SLA)

Die Einhaltung eines Abstraktionsniveaus fördert die Lesbarkeit.

Single Responsibility Principle (SRP)

Fokus erleichtert das Verständnis. Eine Klasse mit genau einer Aufgabe ist verständlicher als ein Gemischtwarenladen.

Separation of Concerns (SoC)

Wenn eine Codeeinheit keine klare Aufgabe hat ist es schwer sie zu verstehen, sie anzuwenden und sie ggf. zu korrigieren oder zu erweitern.

Source Code Konventionen

Code wird häufiger gelesen als geschrieben. Daher sind Konventionen wichtig, die ein schnelles Lesen und Erfassen des Codes unterstützen.

Praktiken

Issue Tracking

Nur, was man aufschreibt, vergisst man nicht und kann man effektiv delegieren und verfolgen.

Automatisierte Integrationstests

Integrationstests stellen sicher dass der Code tut was er soll. Diese wiederkehrende Tätigkeit nicht zu automatisieren wäre Zeitverschwendung.

Lesen, Lesen, Lesen

Lesen bildet!

Gelber Grad

Prinzipien

Interface Segregation Principle (ISP)

Leistungsbeschreibungen, die unabhängig von einer konkreten Erfüllung sind, machen unabhängig.

Dependency Inversion Principle

Punktgenaues Testen setzt Isolation von Klassen voraus. Isolation entsteht, wenn Klassen keine Abhängigkeiten von Implementationen mehr enthalten weder zur Laufzeit, noch zur Übersetzungszeit. Konkrete Abhängigkeiten sollten deshalb so spät wie möglich entschieden werden. Am besten zur Laufzeit.

Liskov Substitution Principle

Wer mit Erben zu tun hat, möchte keine Überraschungen erleben, wenn er mit Erblassern vertraut ist.

Principle of Least Astonishment

Wenn sich eine Komponente überraschenderweise anders verhält als erwartet, wird ihre Anwendung unnötig kompliziert und fehleranfällig.

Information Hiding Principle

Durch das Verbergen von Details in einer Schnittstelle werden die Abhängigkeiten reduziert.

Praktiken

Automatisierte Unit Tests

Nur automatisierte Tests werden auch wirklich konsequent ausgeführt. Je punktgenauer sie Code testen, desto besser.

Mockups (Testattrappen)

Ohne Attrappen keine einfach kontrollierbaren Tests.

Code Coverage Analyse

Traue nur Tests, von denen du weißt, dass sie auch wirklich das Testareal abdecken.

Teilnahme an Fachveranstaltungen

Am besten lernen wir von anderen und in Gemeinschaft.

Komplexe Refaktorisierungen

Es ist nicht möglich, Code direkt in der ultimativen Form zu schreiben.

Grüner Grad

Prinzipien

Open Closed Principle

Weil das Risiko, durch neue Features ein bisher fehlerfreies System zu instabilisieren, so gering wie möglich gehalten werden sollte.

Tell, don’t ask

Hohe Kohäsion und lose Kopplung sind Tugenden. Öffentliche Zustandsdetails einer Klasse widersprechen dem.

Law of Demeter

Abhängigkeiten von Objekten über mehrere Glieder einer Dienstleistungskette hinweg führen zu unschön enger Kopplung.

Praktiken

Continuous Integration

Automatisierung und Zentralisierung der Softwareproduktion machen produktiver und reduzieren das Risiko von Fehlern bei der Auslieferung.

Statische Codeanalyse (Metriken)

Vertrauen ist gut, Kontrolle ist besser und je automatischer, desto leichter ist sie.

Inversion of Control Container

Nur, was nicht fest verdrahtet ist, kann leichter umkonfiguriert werden.

Erfahrung weitergeben

Wer sein Wissen weitergibt, hilft nicht nur anderen, sondern auch sich selbst.

Messen von Fehlern

Nur wer weiß, wie viele Fehler auftreten, kann sein Vorgehen so verändern, dass die Fehlerrate sinkt.

Blauer Grad

Prinzipien

Entwurf und Implementation überlappen nicht

Planungsunterlagen, die mit der Umsetzung nichts mehr gemein haben, schaden mehr, als dass sie nützen. Deshalb nicht die Planung aufgeben, sondern die Chance auf Inkonsistenz minimieren.

Implementation spiegelt Entwurf

Umsetzung, die von der Planung beliebig abweichen kann, führt direkt in die Unwartbarkeit. Umsetzung braucht daher einen durch die Planung vorgegebenen physischen Rahmen.

You Ain't Gonna Need It (YAGNI)

Dinge die niemand braucht, haben keinen Wert. Verschwende an sie also keine Zeit.

Praktiken

Continuous Delivery

Als Clean Code Developer möchte ich sicher sein, dass ein Setup das Produkt korrekt installiert. Wenn ich das erst beim Kunden herausfinde, ist es zu spät.

Iterative Entwicklung

Frei nach von Clausewitz: Kein Entwurf, keine Implementation überlebt den Kontakt mit dem Kunden. Softwareentwicklung tut daher gut daran, ihren Kurs korrigieren zu können.

Komponentenorientierung

Software braucht Black-Box-Bausteine, die sich parallel entwickeln und testen lassen. Das fördert Evolvierbarkeit, Produktivität und Korrektheit.

Test first

Der Kunde ist König und bestimmt die Form einer Dienstleistung. Service-Implementationen sind also nur passgenau, wenn sie durch einen Client getrieben werden.

Weißer Grad

Mit dem weißen Grad schließt sich der Kreis. Er vereinigt alle Prinzipien und Praktiken der farbigen Grade. Sein Horizont sind alle Bausteine zusammen. Wer den weißen Grad erreicht hat, arbeitet ständig an allen Facetten des Wertesystems. Da solche gleichschwebende Aufmerksamkeit jedoch schwer herzustellen ist, erwarten wir, dass der Clean Code Developer nach einiger Zeit wieder mit der Arbeit im Gradesystem von vorne beginnt.