Clean Code Development

https://clean-code-developer.de/

Eine Ausarbeitung und Visualisierung der Prinzipien des Clean Code Development auf Basis der Website clean-dode-developer.de im Ramen des Moduls Verfahren und Werkzeuge moderner Softwareentwicklung im Wintersemester 20/21

Professionalität = Bewustsein + Prinzipien

Prinzipien = Grundlegende Gesetzmäßigkeiten für guten Code
Praktiken = Techniken und Methoden die zum Standard werden

SOLID

S ⇒ Single Responsibility Principle SRP (Eine Klasse soll nur einen Grund für Änderungen haben)
O ⇒ Open Closed Principle OCP (Eine Klasse soll offen für Erweiterungen sein, aber geschlossen gegen Modifikation)
L ⇒ Liskov Substitution Principle LSP (Ein Subtype verhält sich immer wie sein Basetyp)
I ⇒ Interface Segregation Principle ISP (Clients sollen nicht mit Details belastet werden, die sie nicht benötigen)
D ⇒ Dependency Inversion Principle DIP (
- Highlevel Klassen sollen nicht von lowlevel Klassen abhängig sein, sondern beide von Interfaces
- Interfaces sollen nicht von Details abhängig sein, sondern Details vom Interface

)

Wertesystem

Wandelbarkeit

Software soll auch zu einem späteren Zeitpunkt einfach wart- und erweiterbar sein. Bereits bei der Erstellung der Software muss auf eine spätere Erweiterbarkeit hingearbeitet werden. Eine Möglichkeit hierfür ist sich an Regeln und Leitlinien zu orientieren.

Korrektheit

Software sollte bereits mit dem Ziel entwickelt werden, korrekt zu sein. Ein wichtiger Punkt ist die Klarheit über die zu implementierenden Features. Der Auftraggeber hat hier eine Bringschuld, der Entwickler muss jedoch unklare Features hinterfragen. Das Einbeziehen von Tests schützt vor Seiteneffekte bei der Entwicklung.

Produkteffizienz

Das Verhältnis von benötigter Zeit und Produktqualität sollte ausgewogen sein. Dies kann durch Automatisierung, z.B. bei Tests oder Build Vorgängen erreicht werden.

Kontinuierliche Verbesserung

Über das erstellte Produkt und den Weg dorthin sollte reflektiert werden um Potential für Verbesserung zu finden.Der Einsatz von Versionskontrollsysteme und Softwaretests unterstützen dies.

Roter Grad

Unverzichtbare Grundprinzipien, Aufbau einer Grundhaltung

Prinzipien

- **Don't Repeat yourself (DRY)** - Jede Duplizierung im Code kann zu Fehlern führen und sollte vermieden werden. Wird Code mehrmals gebraucht, sollte er in einer Methode ausgelagert werden.
- **Keep it simple, stupid (KISS)** - Code sollte so einfach wie möglich sein. Reviews und Pair Programming sind hier geeignete Mittel zur Umsetzung.
- **Vorsicht vor Optimierungen** - Optimierter Code ist in der Regel schwerer zu lesen, zu verstehen und damit fehleranfälliger und schlechter wartbarer. Die Frage muss immer sein, ob diese Optimierung wirklich notwendig ist.
- **Favour Composition over Inheritance (FCoI)** - Vererbung (**whitebox - reuse**) erzeugt Abhängigkeiten der Children zur Basisklasse. Daher ist die Komposition (**blackbox - reuse**) der bessere Weg:

  Funktionalität wird in Klassen ausgelagert und der Methode übergeben. Als Parameter der Methoden dienen gemeinsame Interfaces.
- **Integration Operation Segregation Principle (IOSP)** - Methoden sollten entweder nur Logik (**Operation**) oder nur Methodenaufrufe (**Integration**) enthalten. Dies führt in der Regel zu wesentlich kürzeren und überschaubaren Methoden, die leichter testbar sind.

Praktiken

- **Die Pfadfinderregel beachten** - Den Code immer etwas besser verlassen, als man ihn vorgefunden hat.
- **Root Cause Analysis** - Bei Problemen versuchen, den ursächlichen Grund zu finden, statt die Symptome zu beheben.
- **Ein Versionskontrollsystem einsetzen** - Ein CVS System sichert den Code und macht ihne wiederherstellbar. Änderungen können gefahrlos umgesetzt werden.
- **Einfache Refaktorisierung Muster einsetzen** - In dieser Stufe ist eines der wichtigsten Muster das Renaming um kryptische Namen zu verbessern.
- **Täglich reflektieren** - Man sollte täglich am Ende des Arbeitstages reflektieren, ob und wie man seine Ziele umgesetzt hat, um einen Abschluß zu finden.

Orangener Grad

Anwendung fundamentaler Grundprinzipien und Automatisierung

Prinzipien

- **Single Level of Abstraction (SLA)** - Innerhalb einer Funktionseinheit sollte der gleiche Abstraktionslevel verwendet werden. Hierbei ist eine Variablenzuweisung weniger abstrakt als ein Methodenaufruf und Methodenaufrufe innerhalb einer Klasse weniger abstrakt als in einem Framework.
- **Single Responsibility Principle (SRP)** - Jede Klasse sollte nur eine Verantwortlichkeit haben um Änderungen auf nur wenige Klassen beschränken zu können. Die Verletzung führt zu hoher Kopplung und größerer Komplexität.
- **Separation of Concerns (SoC)** - Belange sind **total verschiedene Zwecke** (z.B. Logging und Treacking). Sie stehen orthogonal zueinander. In einer Klasse sollten nicht mehrere Belange zusammengefasst werden. Mit Hilfe der aspektorientierten Programmierung können Belange konsequent getrennt werden.
- **Source Code Konvention** - Bezeichner sollten sprechende Namen haben und einem festgelegten Standard folgen.Kommentare im Code sollten sich auf das notwendigste beschränken. Code sollte sich im idealfall selbst beschreiben.

Praktiken

- **Issue Tracking** - Alle offenen Punkte sollten systematisch erfasst und verfolgt werden.
- **Automatisierte Integrationstests** - In laufenden Projekten (**Brownfield Projekte** im Gegensatz zu **Greenfield Projekte**) sollte nach Änderungen systematisch getestet werden, ob alle vorher vorhandenen Funktionalitäten auch später noch wie erwartet funktionieren. Hierfür sollten vor Änderungen automatisierte Integrationstest erstellt werden. Wünschenswert sind auch Unity Tests.
- **Lesen, Lesen, Lesen** - Die ständige und konsequente Weiterentwicklung ist genauso wichtig wie das Üben. Hierzu eignen sich Fachbücher und Blogs.
- **Reviews** - Können als Reviews, Pair Programming, Inspektion, Walkthrough oder technische Reviews umgesetzt werden. Sie helfen, Code zu optimieren und Denkfehler aufzuspüren.

Gelber Grad

Automatisierte Tests und hierfür notwendige Anpassungen im Code durch aufwändigere Refactorings

Prinzipien

- **Interface Segregation Principle (ISP)** - Nicht benötigte Details eines Services sollten nicht angeboten werden, das sie die Abhängigkeit erhöhen. Sein Interface sollte daher nur das notwendigste offenlegen und eine hohe Kohäsion haben. So wird eine enge Kopplung vermieden.
- **Dependency Inversion Principle (DIP)** - High-Level-Klassen sollten nur von einem Interface anhängig sein. Beim Unit Test kann das Low-Level-Objekt so durch ein Mockup ersetzt werden.
- **Liskov Substitution Principle (LSP)** - Das Prinzip besagt, dass sich Subtypen so verhalten müssen wie ihre Basistypen. Subtypen erweitern Basistypen, dürfen sie aber nicht einschränken.
- **Principle of Least Astonishment** - Software sollte überraschungs arm implementiert sein. Namen sollten halten, was sie versprechen. Verstöße führen zu Verwirren und erhöhen die Fehleranfälligkeit.
- **Information Hiding Principle** - Nur das notwendigste öffentlich sein. Ein Verstoß wirkt sich negativ auf die Wandelbarkeit der Software aus.

Praktiken

- **Automatisierte Unit Tests** - Automatsierte Unit Tests decken einzelne Funktionseinheiten wie Klassen, Methoden und Komponenten ab. Bei neuen Code können automatisierte Tests gleich zu Anfang (z.B. Test First Ansatz) implementiert werden.
- **Mockups (Testattrappen)** - Verwenden von Mockups zum Testen, da beim Testen keine Abhängigkeiten vorhanden sein dürfen.
- **Code Coverage Analyse** - Code Coverage Analysen zeigen, welche Codesegmente noch nicht durch Tests abgedeckt worden sind. Unter 90% sollte die Testabdeckung nicht fallen. Man unterscheidet C0 und C1 Kennzahlen.

  - `C0 = (getestete Anweisungen / gesamten Anweisungen) * 100%`
  - `C1 = (getestete Entscheidungen o. Zweige / gesamte Entscheidungen o. Zweige) * 100%`

  C1 ist dabei die stärkere Kennzahl. C1 impliziert C0, nicht jedoch nicht umgekehrt.
- **Teilnahmen an Fachveranstaltungen** - Man sollte Kontakt zu anderen Entwicklern suchen. Im Betrieb, lokal und überregional.
- **Komplexe Refaktorisierung** - benötigen automatisierte Tests um sicherzustellen, dass Software wie erwartet funktioniert.

Grüner Grad

Weiterführung der automatisierten Tests

Prinzipien

- **Open Closed Principle (OCP)** - Eine Klasse soll offen für Erweiterungen, aber geschlossen für Modifikation sein soll. Dies kann erreicht werden, indem konkrete Berechnungen über ein Interface in eine andere Klasse ausgelagert wird (**Strategy Pattern**).
- **Tell, don't ask** - Klassen sollten nicht zu viel von sich preisgeben. Im idealfall existieren nur Methodenaufrufe mit dem ein Objekt aufgefordert wird, etwas zu tun. Die Umsetzung obliegt dem Objekt.
- **Law of Demeter** - Zuviel Zusammenarbeit von Objekten kann zu unschön enger Kopplung führen. Daher sollte dem Prinzip "Don't talk to strangers" gefolgt werden. Eine Methode sollte im idealfall nur Methoden
  - der eigenen Klassen
  - der Parameter
  - der assoziierten Klassen und
  - selbst erzeugter Objekte nutzen.

Praktiken

- **Continuous Integration** - Code sollte zu jeder Zeit vollständig lauffähig sein. Ein Werkzeug hierfür ist Continuous Integration. Es sorgt dafür, das Code nach jeder Übermittlung von Änderungen automatisiert getestet und compiliert wird. Hierbei wird nicht nur der geänderte Code berücksichtigt. Trotzdem sollten Tests vor jeder Übermittlung von Änderungen durch den Entwickler auch lokal ausgeführt worden sein. Fehler können so sehr früh identifiziert werden.
- **Statische Codeanalyse (Metriken)** - Software soll wandelbar und lange verwendbar sein. Gleichzeitig steigt die Komplexität von Software. Metriken können Schlüsselwerte ermitteln, mit denen Software beurteilt werden kann. Tools für diesen Zweck sollten immer eingesetzt werden.
- **Inversion of Control Container** - Abhängigkeiten sollten automatisiert aufgelöst werden. Hierzu stehen zwei Verfahren zur Verfügung. Beide verwenden einen Inversion of Control Container (IoC Container) in dem die verwendeten Klassen zuvor hinzugefügt werden müssen. Es existieren die zwei Verfahren Locator und Container. Beim Locator müssen die Klassen manuell hinzugefügt werden.
- **Erfahrung Weitergeben** - Eigene Erfahrung sollte weiter gegeben werden, fremde Erfahrung aufgenommen und bedacht werden.
- **Messen von Fehlern** - Fehler passieren und sind wichtig, um sich und eine Software zu verbessern. Schlecht sind Fehler, welche von außen zurück gemeldet werden, da diese nicht im Entwicklungsstadium gefunden worden sind.

Blauer Grad

Techniken und Methoden die zum Standard werden

Prinzipien

- **Entwurf und Implementation überlappen nicht** - Entwurf und Implementierung müssen übereinstimmen. Daher sollte Architektur und Implementation strikt getrennt sein. Aufgabe der Architektur ist es, Software in Komponenten zu zerlegen, deren Abhängigkeiten zu definieren und Leistungen in Kontrakten zu beschreiben. Aufgabe der Implementierung ist, die Komponenten zu realisieren. die innerer Struktur ist für die Architektur unsichtbar.
- **Implementation spiegelt Entwurf** - Die Implementierung sollte den Entwurf auch physisch widerspiegeln. Unbeabsichtigte Änderungen der Architektur sollten unmöglich sein.
- **You Ain't Gonna Need it (YAGNI)** - Nur was wirklich gebraucht wird, darf implementiert werden. Hat man Zweifel, ob sein ein Aufwand lohnt, sollte man sich dagegen entscheiden oder die Entscheidung vertagen und sich so spät wie möglich entscheiden.
  - nur klare Anforderungen werden implementiert
  - Anforderungen werden vom Kunden priorisiert
  - Umsetzung erfolgt nach Priorität
  - Vorsorge für Änderungen und Erweiterungen (Entwicklungsprozess und Codestruktur)

Praktiken

- **Continuous Delivery** - Erweitert den Prozess des Continuous Integration und automatisiert das Setup und Deployment.
- **Iterative Entwicklung** - Projekte verlaufen (außer sehr kleine) in mehreren Iterationen. In jeder Iteration sollte man sich auf ein Subset von Forderungen konzentrieren, welche in 2 bis 4 Wochen auslieferbar sind, um ggf. nicht zuviel Zeit verloren zu haben. Wichtig ist, das der Kunde immer eine lauffähige Version hat. Hierzu gehört auch ein Setup und die Durchführung aller Tests. Der Entwicklungsablauf muss diese Iterationen berücksichtigen. Hierzu gehört auch eine Rückkehr zu einem früheren Stand (Versionskontrollsystem).
- **Komponentenorientierung** - Es wird mehr Wert auf die Strukturierung anhand von Assemblies gelegt. Ein Weg zur unabhängigen Programmierung von Assemblies sind separate Visual Studio Solutions je Assembly
- **Test First** - An erster Stelle steht der Test, dann die Implementierung. Der Test testet den Code und dokumentiert ihn aber auch gleichzeitig und verleiht Schnittstellen eine Semantik.

Weißer Grad

Berücksichtigung aller Prinzipien und Praktiken. Nachdem man in Folge 21 Tage alle Prinzipien verfolgt und umgesetzt hat, hat man den weißen Grad erreicht. Jetzt fängt man wieder von vorne an, wiederholt und vertieft jedoch die Prinzipien.