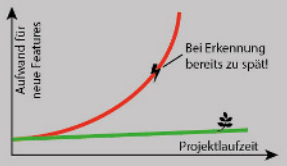
Clean Code Developer Prinzipien und Praktiken für mehr Softwarequalität

Wertesystem

Das Wertesystem leitet Clean Code Developer in ihrer täglichen Arbeit. Es enthält keine Problemlösungen, sondern definiert Rahmenbedingungen für Problemlösungen.

Evolvierbarkeit

Damit Änderungen möglich sind, muss die Software eine innere Struktur haben, die solche Änderungen begünstigt. Die Evolvierbarkeit ist ein Kriterium bei der Entwicklung von Software, das anzeigt, mit welcher Energie und welchem Erfolg neue Features eingebracht werden können. Sie kann nicht nachträglich hinzugefügt werden, sondern muss von vorneherein berücksichtigt werden.



Korrektheit

Korrektheit muss bereits während der Entwicklung berücksichtigt werden, nicht nur einmalig nach ihrer Fertigstellung. Dafür müssen die Entwickler die Anforderungen kennen und verstehen. Bei Unklarheiten müssen sie ggf. nachfragen.

Produktionseffizienz

Manuelle Arbeitsschritte benötigen viel Zeit und sind fehleranfällig. Das führt zu langen Entwicklungszeiten und hohen Fehlerraten, deren Behebung weiteren Aufwand verursacht, der wiederum die Kosten steigen lässt. Die Produktionseffizienz ist ebenso wichtig, um die anderen Werte in ein maßvolles Verhältnis zu setzen. Wer unendlich viel Aufwand für die Korrektheit treibt, macht am Ende auch etwas falsch.

Reflexion

Ohne Rückschau ist keine Weiterentwicklung möglich. Nur wer reflektiert, wie er eine Aufgabenstellung gelöst hat, kann feststellen, ob der gewählte Weg einfach oder beschwerlich war. Lernen basiert auf Reflexion.

Prinzipien und Praktiken

Prinzipien

Grundlegende Gesetzmäßigkeiten für die Strukturierung von Software. Code sollte immer im Einklang mit einer maximalen Zahl von Prinzipien sein. Die Nicht-Einhaltung eines Prinzips führt kurz- bis mittelfristig zu geringerem Code-Verständnis oder höherem Aufwand für Änderungen.

Ob ein Prinzip eingehalten wurde, kann man dem Code immer ansehen.

Praktiken

Techniken und Methoden, die ständig zum Einsatz kommen. Es sind handfeste Handlungsanweisungen, die manchmal des Einsatzes von Werkzeugen bedürfen.

Ob einer Praktik gefolgt wird, kann man dem Code nicht immer ansehen.

3. Prinzipien

Schnittstellenaufteilungsprinzip (ISP)

- Schnittstellen sollten eine hohe Kohäsion haben, also nur Dinge enthalten, die wirklich eng zusammengehören
- Schlanke Schnittstellen erfordern weniger Methodenimplementierungen
- Code wird besser wartbar, kompakter, besser wiederverwertbar und besser überprüfbar
- Refaktorisierungsmuster: Schnittstelle extrahieren, Basisklasse extrahieren

Abhängigkeits-Umkehrungs-Prinzip (DIP)

- High-Level-Klassen sollen nicht von Low-Level-Klassen abhängig sein, sondern beide von Schnittstellen
- Schnittstellen sollen nicht von Details abhängig sein, sondern Details von Schnittstellen
- Ermöglicht isoliertes Testen einzelner Klassen:
  Konkrete Implementierungen können im Test durch Mockups ersetzt werden
- Injizierung der Abhängigkeiten zunächst nur mit Konstruktor-Parametern (↗ IoC-Container)

Liskovsches Substitutionsprinzip (LSP), Ersetzbarkeitsprinzip

- Subtypen müssen sich so verhalten wie ihr Basistyp
- Subtypen dürfen die Funktionalität eines Basistyps nur erweitern, nicht einschränken
  (z.B. durch eingeschränkten Wertebereich oder zusätzliche Ausnahmefehler)
- Betrachtung der Ableitung als „verhält-sich-wie-Relation" statt nur „ist-ein-Relation („Ein Kreis ist keine Ellipse")
- Sehr genau über Vererbung nachdenken, oft ist Komposition besser (↗ PCoC)

Prinzip der geringsten Überraschung

- Wenn sich eine Komponente überraschenderweise anders verhält als erwartet, wird ihre Anwendung unnötig kompliziert und fehleranfällig
- Abfragemethoden wie GetValue() sollen den Zustand nicht verändern

Information-Hiding-Prinzip

- Verbergen von Details in einer Schnittstelle reduziert Abhängigkeiten
- Mehr öffentlich sichtbare Details erhöhen die Kopplung zwischen der Klasse und ihren Verwendern
- Sobald ein ein Detail benutzt wird, lässt es sich schwerer wieder ändern

3. Praktiken

Automatisierte Unit-Tests

- Test einzelner Klassen oder Methoden
- Dafür müssen Funktionseinheiten von ihren Abhängigkeiten befreit werden können
- Evtl. erforderliche Refaktorisierungen werden durch vorhandene Integrationstests abgedeckt
- Automatisierung spart Zeit (Testfälle werden immer mehr) und nimmt Angst vor Fehlern bei der Durchführung

Testattrappen (Mockups)

- Um eine Komponente isoliert zu testen, müssen die Abhängigkeiten abgetrennt werden
- Attrappen ersetzen die Abhängigkeiten
- Zu testende Komponente interagiert mit gut kontrollierbaren Attrappen statt echten Komponenten

Code-Abdeckungsanalyse

- Unit-Tests sollten möglichst alle Pfade durch den Code abdecken
- Über nicht getestete Code-Abschnitte kann keine Korrektheitsaussage gemacht werden
- Code-Abdeckungsanalyse findet Anweisungen oder Entscheidungen, die nicht getestet werden
- Mindestens 90 % erforderlich, maximal 100 % anstrebenswert

Komplexe Refaktorisierungen

- Es ist nicht möglich, Code direkt in der ultimativen Form zu schreiben
- Refaktorisierungen, die über die im roten Gang genannten hinausgehen, werden durch Tests überprüft

Teilnahme an Fachveranstaltungen

- Am besten lernen wir von anderen und in Gemeinschaft
- Austausch mit Entwicklern außerhalb des eigenen Teams, um andere Meinungen zu erfahren

2. Prinzipien

Single Level of Abstraction (SLA)

- Einhaltung eines einzelnen Abstraktionsniveaus pro Code-Abschnitt fördert die Lesbarkeit
- High-Level-Funktionen vor Low-Level-Operationen trennen
- Nicht in einer Funktion vermischen
- Leser kann sich so einen Überblick über eine Klasse verschaffen und nur bei Bedarf die Implementierungsdetails nachlesen

Trennung der Belange (SoC, SRP)

- Verschiedene Elemente der Aufgabe sollten möglichst in verschiedenen Elementen der Lösung repräsentiert werden
- Anpassungen sind dadurch lokal begrenzt und überschaubar
- Nebenwirkungen auf andere Funktionen werden vermieden
- Einzelne Belange werden isoliert testbar (↗ Unit tests)

Quelltextkonventionen

- Code wird häufiger gelesen als geschrieben
- Konventionen sind wichtig, unterstützen schnelles Lesen und Erfassen des Codes
  - Namensregeln machen Code schneller und besser verständlich
  - Kommentare sollten keine Defizite im Code-Stil ausgleichen:
    Code sollte so klar und deutlich sein, dass er möglichst ohne Kommentare auskommt

2. Praktiken

Fehlerverfolgung

- Probleme, offene Punkte und Wünsche strukturiert erfassen und aufschreiben
- Aufgaben werden nicht vergessen und können effektiv delegiert und nachverfolgt werden

Automatisierte Integrationstests

- Integrationstests stellen sicher, dass sich die Anwendung nach einer Änderung (Refaktorisierung oder Erweiterung) noch so verhält wie vorher
- Automatisierung ist für Effizienz erforderlich
- Unit-Tests folgen später, wenn der Code dafür vorbereitet ist (↗ SRP)

Reviews

- Vier Augen sehen mehr als zwei
- Alleine das Erklären des eigenen Codes führt manchmal zu besserem Verständnis
- Permanente informelle Reviews durch Pair Programming oder formelle Code-Reviews
- Bereits sehr früh im Entwicklungsprozess und auch für Anforderungen möglich
- Je früher Fehler gefunden werden, desto günstiger ist deren Beseitigung

Lesen, Lesen, Lesen

- Softwaretechnik, Methoden und Werkzeuge entwickeln sich ständig weiter
- Regelmäßig Fachpublikationen lesen (Bücher, Zeitschriften, Blogs, Videos)
- Auf dem Laufenden bleiben
- Fähigkeit, informierte Entscheidungen zu treffen

4. Prinzipien

Offen-Geschlossen-Prinzip (OCP)

- Module sollten offen für Erweiterungen und geschlossen für Modifikationen sein
- Veränderung existierenden Codes bei Erweiterung vermeiden (Fehlerquelle)
- Erweiterung durch Strategie-Entwurfsmuster oder Ableitung

Tell, don't ask

- Methoden mitteilen, was sie tun sollen, statt nach ihrem internen Zustand zu fragen und selbst zu entscheiden (↗ Information Hiding)
- Fördert Kohäsion und lose Kopplung

Gesetz von Demeter

- Abhängigkeiten von Objekten über mehrere Glieder erhöhen die Kopplung
- Nur nahe Aufrufe gestattet:
  - Methoden der eigenen Klasse
  - Methoden der Parameter
  - Methoden assoziierter Klassen
  - Methoden selbst erzeugter Objekte
- Ausnahme: Reine Datenhaltungsklassen

4. Praktiken

Continuous Integration (CI)

- Übersetzung und Test erfolgt automatisch bei jedem Commit in der Versionsverwaltung
- Übersetzung in einer neutralen Umgebung ohne lokale Anpassungen
- Fehler in anderen Codebereichen werden frühzeitig erkannt
- Automatisierung beschleunigt den Vorgang und vermeidet Fehler, die bei manueller Arbeit unter Zeitdruck entstehen

Statische Code-Analyse (Metriken)

- Anforderungskonformität als Minimum ist selbstverständlich
- Korrektheit wird durch automatisierte Tests überprüft
- Metriken über die Evolvierbarkeit können teilweise durch Tools berechnet werden

Inversion-of-Control-Container (IoC)

- Unterstützung bei Umsetzung des Abhängigkeits-Umkehrungs-Prinzip (DIP)
- Trennung der Belange (SoC) führt zu vielen kleineren Klassen
- Vereinfachen die Umkonfiguration der Klassen für Testfälle, z.B. durch Testattrappen

Messen von Fehlern

- Zur Erkennung einer Verbesserung müssen die Fehler gemessen werden
- Relevant sind Fehler, die vom Kunden nach einer Iteration gemeldet werden
- Messung durch Zählen oder Zeitnahme
- Vergleichbarkeit der Messung wichtiger als Präzision

Erfahrung weitergeben

- Lernen zur eigenen Anwendung ist durch den konkreten Einsatzzweck begrenzt
- Wissensvermittlung erfordert ein tieferes Verständnis des Themas
- Übung durch Präsentationen vor realem Publikum (unmittelbares Feedback) oder durch Veröffentlichung von Texten (Fachzeitschriften, Blog)

5. Prinzipien

Entwurf und Implementierung überlappen nicht

- Entwurf/Architektur zerlegt Software in Komponenten, definiert Abhängigkeiten und Kontrakte
- Implementierung liegt den internen Aufbau von Komponenten fest
- Trennung der Zuständigkeiten vermeidet Wiederholungen (DRY) und damit Inkonsistenzen
- Architekten sehen Komponenten als Black Boxes
- Komponentenimplementierer kennen nur die Kontrakte, größerer Zusammenhang nicht erforderlich

Implementierung spiegelt Entwurf

- In der Architektur definierte Komponenten auch im Code möglichst physisch trennen
- Verbesserung der Übersichtlichkeit und Testbarkeit
- Beiläufige Architekturänderungen während der Implementierung sollten unmöglich sein
- Erkenntnisse während der Implementierung dürfen aber auf die Planung zurückwirken

Du wirst es nicht brauchen (YAGNI)

- Funktionalität soll erst dann implementiert werden, wenn klar ist, dass sie tatsächlich gebraucht wird
- Anforderungen sind oft ungenau
- Umsetzung (noch) nicht geforderter Funktionalitäten bindet Ressourcen, verzögert wichtigere Arbeiten und könnte sich später als hinderlich herausstellen
- Wenn im Zweifel, entscheide dich gegen den Aufwand

5. Praktiken

Continuous Delivery (CD)

- Erweiterung der Continuous Integration (CI)
- Testet auch die Installierbarkeit des Produkts in einer neutralen Umgebung
- Befähigt jedes Teammitglied, jederzeit zur Erstellung eines installationsfertigen Produkts

Iterative Entwicklung

- Um Feedback aus der Implementierung oder dem Kundentest nutzen zu können, muss der Entwicklungsprozess Schleifen enthalten
- Mindestens Schleife vom Kundentest zurück zur Planung notwendig
- Kundenanforderungen werden stappenweise umgesetzt
- Ziel jeder Iteration festlegen: auslieferungsfertige, getestete Software
- Kundenfeedback alle 2. bis 4 Wochen vermeidet lange Irrwege
- Retrospektive des Teams nach jeder Iteration zur organisatorischen Entwicklung und zur Verbesserung von Schätzungen

Komponentenorientierung

- Anwendungsprozess besteht aus Komponenten, die bestehen aus Klassen
- Stärkere Gliederung der Bereiche einer Anwendung
- Verbesserung der Übersichtlichkeit
- Lose Kopplung der Komponenten durch vorher definierte Kontrakte
- Parallele Implementierung möglich

Test zuerst

- Erstellung der Spezifikation von außen nach innen, beginnend beim Anwender
- Dadurch wird nur das spezifiziert, was letztlich benötigt wird (↗ YAGNI)
- Beschreibung der Schnittstellen sowie des gewünschten Verhaltens durch Tests
- Tests sind gleichzeitig Spezifikationsdokumentation, separate Dokumentation nicht mehr nötig (DRY)
- Spezifikation ist kein passiver Text sondern ausführbarer Code und kann gleich automatisch geprüft werden
- Qualitätssicherung beginnt vor der Implementierung

1. Prinzipien

Wiederhole dich nicht (DRY)

- Doppelung von Code/Handgriffen begünstigt Inkonsistenzen/Fehler
- Erkennen von sich wiederholendem Code oder anderen Artefakten, die man selbst oder andere erzeugt haben
- Beseitigen durch Refaktorisierungen
- Auch unter Zeitdruck

Halte es einfach (KISS)

- Wer mehr tut als das Einfachste, lässt den Kunden warten und macht die Lösung unnötig kompliziert
- Für die Evolvierbarkeit muss Code verständlich sein
- Reviews und Pair Programming zur Kontrolle

Vorsicht vor Optimierungen

- Optimierungen kosten Aufwand und verringern die Code-Lesbarkeit
- Sie sind oft nicht notwendig oder nützlich (↗ YAGNI)
- Wenn, dann nur nach Analyse mit einem Profiler

Bevorzuge Komposition über Ableitung (PCoC)

- Komposition fördert lose Kopplung und Testbarkeit und ist oft flexibler
- Eine Klasse verwendet die andere, unter Nutzung von Schnittstellen
- Konkrete Implementierungen werden austauschbar
- Bei Ableitung ist eine Klasse von Implementierungsdetails der Basisklasse abhängig

1. Praktiken

Hinterlasse einen Ort immer in einem besseren Zustand, als du ihn vorgefunden hast (Pfadfinderregel)

- Beim Bearbeiten von Code auch gleich auffällige Unklarheiten verbessern
- Konsequentes Beheben von Problemen, bevor sie vergessen werden oder sich vergrößern

Ursachenanalyse

- Symptome behandeln bringt schnell Linderung, kostet langfristig aber mehr Aufwand
- Unter die Oberfläche von Problemen schauen ist letztlich effizienter

Versionsverwaltungssystem einsetzen

- Angst vor Beschädigung eines laufenden Systems lähmt die Softwareentwicklung
- Mit einem Versionsverwaltungssystem ist solche Angst unbegründet
- Entwicklung kann schnell und mutig voranschreiten

Refaktorisierungsmuster „Methode extrahieren" und „Umbenennen" einsetzen

- Kenntnis typischer Verbesserungshandgriffe erleichtert die Verbesserung des Codes
- Durch „Methode extrahieren" wird sich wiederholender Code zusammengefasst
- Durch „Umbenennen" werden unverständliche Namen verbessert

Täglich reflektieren

- Keine Verbesserung, kein Fortschritt, kein Lernen ohne Reflexion
- Nur, wenn Reflexion eingeplant wird, findet sie unter dem Druck des Tagesgeschäfts auch statt

Die fünf Grade

Clean Code Developer *ist* man nicht einfach, sondern man *wird* es. Es braucht Zeit und Übung, das Wertesystem zu verinnerlichen. Die Aufteilung in Stufen erleichtert den Einstieg. Ein Grad ist nicht „besser" als ein anderer. Sie bauen aufeinander auf und erleichtern das Erlernen. Grade sind Aspekte auf das Große Ganze, sie werden immer wieder aufs neue durchlaufen. Letztlich sind alle Grade erforderlich.

0. Schwarzer Grad

- An Clean Code Development interessiert
- Sucht noch den Einstieg oder organisatorische Hürden stehen im Weg
- Arbeitet noch nicht am ersten Grad im Sinne des CCD-Wertesystems

1. Roter Grad

- Leichter Einstieg in die Übungspraxis
- Enthält nur Elemente, die unverzichtbar sind
- Aufbau einer fundamentalen Haltung zur Softwareentwicklung und zum Clean Code Developer

2. Oranger Grad

- Anwendung einiger fundamentaler Prinzipien auf den Code
- Produktivitätssteigerung durch Automatisierung von Abläufen

3. Gelber Grad

- Detaillierte automatisierte Tests
- Änderung der Codierungspraxis für das isolierte Testen einzelner Module

4. Grüner Grad

- Weitere Automatisierung
- Zentrale Code-Erstellung und Tests

5. Blauer Grad

- Automatisierte Auslieferung
- Planung der Architektur
- Iteratives Vorgehensmodell

6. Weißer Grad

- Führt alle Prinzipien, Regeln und Praktiken der anderen Grade zusammen
- Lernen passiert in Schleifen und braucht Wiederholung
- Ständige Iterationen des Kreislaufs dienen der Verfeinerung der Anwendung der Aspekte der einzelnen Grade
- Erfordert mehrere Jahre Erfahrung und eine geeignete Umgebung

Weitere Erklärungen und Anleitung auf der Projektwebsite im Internet:

http://clean-code-developer.de