# Clean Code Developer

1. Roter Grad
   * DRY- Don’t Repeat Yourself
     + Jede Doppelung von Code oder auch nur Handgriffen leistet Inkonsistenzen und Fehlern Vorschub
   * KISS – Keep it simple, stupid
     + Wer mehr tut als das Einfachste, lässt den Kunden warten und macht die Lösung unnötig kompliziert
     + Einfache, klare und leicht verständliche Lösung
     + Andere Entwickler sollen Code schnell verstehen können
     + Regelmäßige Reviews und Pair Programming, zur Überprüfung, ob die einfachste Lösung verwendet wurde
   * Vorsicht vor Optimierungen!
     + Optimierungen kosten immer viel Aufwand. Wer Vorsicht walten lässt, spart oft wertvolle Ressourcen für das, was dem Kunden wirklich nützt
     + Verständlichkeit und Evolvierbarkeit hat Priorität! Durch Optimierung wird diese meist verschlechtert
   * FCol – Favour Composition over Inheritance
     + Komposition fördert die lose Kopplung und die Testbarkeit eines Systems und ist oft flexibler
     + Bei der Komposition verwendet eine Klasse eine andere🡪 klar definierte Schnittstelle fördert die Entkoppelung + verschiedene Implementationen einfach ausgetauscht
     + Frage stellen, ob man Komposition nicht Vorrang gibt, statt Vererbung (meist unnötig komplex, schlechter testbar etc.)
   * IOSP – Integration Operation Segregation Principle
     + Ein deutliches Symptom schlecht wandelbaren Codes sind tiefe Hierarchien funktionaler Abhängigkeit. Sie reduzieren die Verständlichkeit und erschweren automatisierte Tests wie Refactoring
     + Kurze Methodenaufrufe 🡪 sind leicht zu testen, da keine Abhängigkeiten + leicht zu verstehen etc.

* Praktiken
  + Pfadfinderregel
    - Hinterlasse einen Ort immer in einem besseren Zustand als du ihn vorgefunden hast
  + 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
  + Versionskontrollsystem einsetzen
    - Angst vor Beschädigung eines “running system” lähmt die Softwareentwicklung. Mit einer Versionsverwaltung ist solche Angst unbegründet. Die Entwicklung kann schnell und mutig voranschreiten.
  + 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.
    - Methode extrahieren nach DRY-Prinzip, Umbenennen (keine kryptischen Namen), Werkzeugunterstützung durhc IDEs oder Werkzeuglisten
  + 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.
    - Kleinschrittige Planung + Reflexion nach jedem Schritt

1. Orangener Grad

* SLA – Single Level of Abstraction
  + Die Einhaltung eines Abstraktionsniveaus fördert die Lesbarkeit 🡪 damit Code gut zu lesen/zu verstehen sollten Methoden nur ein Abstraktionsniveau verwenden
* SRP – Single Responsibility Principle (SOLID PRINZIP)
  + Fokus erleichtert das Verständnis. Eine Klasse mit genau einer Aufgabe ist verständlicher als ein Gemischtwarenladen –> eine Klasse sollte nur eine Verantwortlichkeit haben
  + Änderungen oder Erweiterungen der Funktionalität einer Anwendung sollen sich auf wenige Klassen beschränken
* SoC – Separation of Concerns
  + Wenn eine Codeeinheit keine klare Aufgabe hat ist es schwer sie zu verstehen, sie anzuwenden und sie ggf. zu korrigieren oder zu erweitern
  + In jeder Responsibility können zwar mehrere Concerns enthalten sein, aber getrennt z.B. durch Gruppierung
* Source Code Konventionen
  + Code wird häufiger gelesen als geschrieben. Daher sind Konventionen wichtig, die ein schnelles Lesen und Erfassen des Codes unterstützen
    - Namensregeln
      * Ohne Namensregeln muss man sich wieder und wieder auf den Stil einzelner Entwickler einstimmen 🡪 auf Variante einigen
    - Richtig kommentieren
      * Unnötige oder gar falsche Kommentare halten beim Lesen auf. Der Code sollte so klar und deutlich sein, dass er möglichst ohne Kommentare auskommt
      * Kommentar Hinweis darauf, dass Code verbessert werden kann
      * Bedarf an Kommentaren reduzieren, in dem man gute Namen verwendet für Variablen etc.
* 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
    - Nach Änderungen im Code sollten wir sicher sein, dass wir dabei nichts kaputt machen 🡪 Sicherheit durch diese Tests
    - Diese Tests werden ganz oben auf Benutzerschnittstelle oder weiter unten gesetzt (ganz oben = testet durch alle Layer)
    - Unit Tests auch wichtig 🡪 Code muss SRP Prinzip erfüllen, damit einzelne Funktionseinheiten isoliert getestet werden können
  + Lesen
    - Lesen und Recherche bildet
  + Reviews
    - Vier Augen sehen mehr als zwei. Wenn der eine Entwickler dem anderen seinen Code erklärt, tauchen meist Details auf, die bislang nicht bedacht wurden

1. Gelber Grad

* ISP – Interface Segregation Principle (SOLID PRINZIP)
  + - Leistungsbeschreibungen, die unabhängig von einer konkreten Erfüllung sind, machen unabhängig
    - Client nicht von Details eines Service abhängig, die er gar nicht benötigt
    - Je weniger in Interface enthalten ist, desto geringer Kopplung zwischen den beiden Komponenten 🡪 nur Dinge enthalten, die wirklich eng zusammen gehören
  + Dependency Inversion Principle (SOLID PRINZIP)
    - 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
    - High-Level Klassen sollen nicht von Low-Level Klassen abhängig sein, sondern beide von Interfaces
    - Interfaces sollen nicht von Details abhängig sein, sondern Details von Interfaces
    - Auflösung abstrakte Abhängigkeit mit Objekt 3 Möglichkeiten 🡪 mit Konstruktorparameter „per Hand“, Einsatz eines IoC Container, Dependency Lookup
  + Liskov Substitution Principle (SOLID PRINZIP)
    - Subtyp erweitert die Funktionalität eines Basistyps, darf aber nicht einschränken
    - Wenn Methode im Basistyp auf bestimmten Wertebereich definiert ist, darf der Subtyp Wertebereich übernehmen/erweitern, aber nicht einschränken
    - Über Vererbung sehr genau nachdenken, da Komposition in meisten Fällen vorzuziehen
  + Principle of Least Astonishment
    - Wenn sich eine Komponente überraschenderweise anders verhält als erwartet, wird ihre Anwendung unnötig kompliziert und fehleranfällig
    - Software sollte überraschungsarm implementiert sein z.B. Methode mit 2 Bedeutungen, versucht man zu meiden, da man Überraschung rechnen kann
    - Testgetriebene Entwicklung fördert überraschungsarme Schnittstellen, da die Schnittstelle aus der Sichtweise ihrer Verwendung entworfen + implementiert wird
  + Information Hiding Principle
    - Durch das Verbergen von Details in einer Schnittstelle werden die Abhängigkeiten reduziert 🡪 je mehr Details von außen sichtbar, desto höher Kopplung zwischen Klasse und ihren Verwendern 🡺 benutzt Verwender erstmal ein Detail ist es schwer dies zu verändern
* 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
    - Oft größere Eingriffe als nur Umbenennen und Methode extrahieren 🡪 funktioniert nur mit vorhandenen automatisierten Tests effizient und risikolos

1. Grüner Grad

* Open Closed Principle (SOLID PRINZIP)
  + Klasse soll offen für Erweiterungen sein, jedoch geschlossen gegenüber Modifikationen
* Tell, don’t ask
  + Hohe Kohäsion und lose Kopplung sind Tugenden. Öffentliche Zustandsdetails einer Klasse widersprechen dem
  + Klassen sollen keine Property Getter haben 🡪 statt dem Objekt mitzuteilen, was es tun soll, wird es befragt, um dann von außen Betrachtungen über den internen Zustand des Objektes anzustellen 🡪 Entstehung von Objekten mit Verhalten statt dummer Datenhaltungsobjekte
* Law of Demeter
  + Abhängigkeiten von Objekten über mehrere Glieder einer Dienstleistungskette hinweg führen zu unschön enger Kopplung
  + Zusammenspiel von Objekten auf ein gesundes Maß zu beschränken
  + Eine Methode soll nur folgende andere Methoden verwenden:
    - Methoden
      * Der eigenen Klasse
      * Der Parameter
      * Assoziierter Klassen
      * Selbst erzeugter Objekte
* 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.
    - Tools für:
      * Legacy 🡪können die Tools den Status Quo erheben und somit eine Grundlinie definieren, mit der die weitere Entwicklung des Codes (zum Besseren) verglichen werden kann
      * Neuen Code 🡪der mit Evolvierbarkeit im Blick geplant wurde, zeigt solch statische Codeanalyse, ob er das Ideal der Planung erfüllt.
  + Inversion of Control Container
    - Automatisierung Auflösen von Abhängigkeiten
    - Klassen im Container hinterlegen, dann kann Container Instanzen der hinterlegten Klasse liefern
      * Mit Locator geschieht explizit
      * Bei Logging übliches Verfahren 🡪 Abhängigkeiten als Parameter des Konstruktors aufgeführt + sichtbar 🡪 Container in der Lage Abhängigkeiten implizit aufzulösen, indem er rekursiv alle nötigen Objekte über Container instanziert etc.
  + 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.

1. Blauer Grad

* 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.
  + Entwurf und Planung müssen dem DRY-Prinzip entsprechen
    - So wenig wie möglich überlappen 🡪 stellen dann keine Wiederholungen mehr dar, sondern beschreiben unterschiedliches
  + Aufgabe Architektur 🡪 Software in Komponenten zerlegen, deren Abhängigkeiten zu definieren und Leistungen in Kontrakten zu beschreiben ( Strukturen durch Architekten gepflegt)
  + Aufgabe Implementation 🡪 von Architektur definierten Komponenten zu realisieren (nicht architekturrelevant + innere Struktur für Architektur unsichtbar)
* 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
  + Architektur und Implementation sollen nicht überlappen, damit DRY-Prinzip nicht verletzen 🡪 Inkonsistenzen können vermieden werden, die dadurch entstehen können, dass auf der einen Seite etwas geändert wird, ohne diese Änderung auf der anderen Seite nachzuführen
  + Planung muss sein und Implementation darf Planung nicht torpedieren 🡪 deshalb sind Planung und Implementation zu entkoppeln
    - Wo das nicht möglich ist, sollte Planung mit Mitteln der Implementation arbeiten und die Implementation physisch die Planung widerspiegeln
* You Ain’t Gonna Need It (YAGNI)
  + Dinge die niemand braucht, haben keinen Wert. Verschwende an sie also keine Zeit
  + Wer sie anwendet, schneidet ein Problem in keine Würfel des unmittelbar Nötigen 🡪 nur das unzweifelhaft und unmittelbar Nutzbringende implementiert
  + Im Zweifel entscheide dich gegen den Aufwand
  + Bedeutung für die Softwareentwickung 🡪 klare Anforderungen implementieren, priorisiert klare Anforderungen, Umsetzen der klaren Anforderungen in der Reihenfolge ihrer Priorisierung + Entwicklungsprozess + Codestruktur im Großen und Kleinen so aufsetzen, dass keine Angst aufkommt, sich ändernde und neue Anforderungen zu realisieren
  + YAGNI ist immer in Anschlag zu bringen genauso wie DRY
* 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
    - Service-Implementationen sind also nur passgenau, wenn sie durch einen Client getrieben werden