# Clean-Coder

# Was ist ein guter Entwickler?

Verhaltensregeln für professionelle Programmierer

# ➔ Wer bin ich?

**André Hänsel** - Dipl. Informatiker (FH) - Seit 2010 als Entwickler tätig.

**Seit 2015:** move:elevator - Business Solution Team
**Davor:** Ressourcenmangel Dresden, Net-Clipping, Seto GmbH

„Das dauert doch nur 5 Minuten.“

„Geht das auch in zwei Stunden?“

„Das haben wir schon verkauft.“

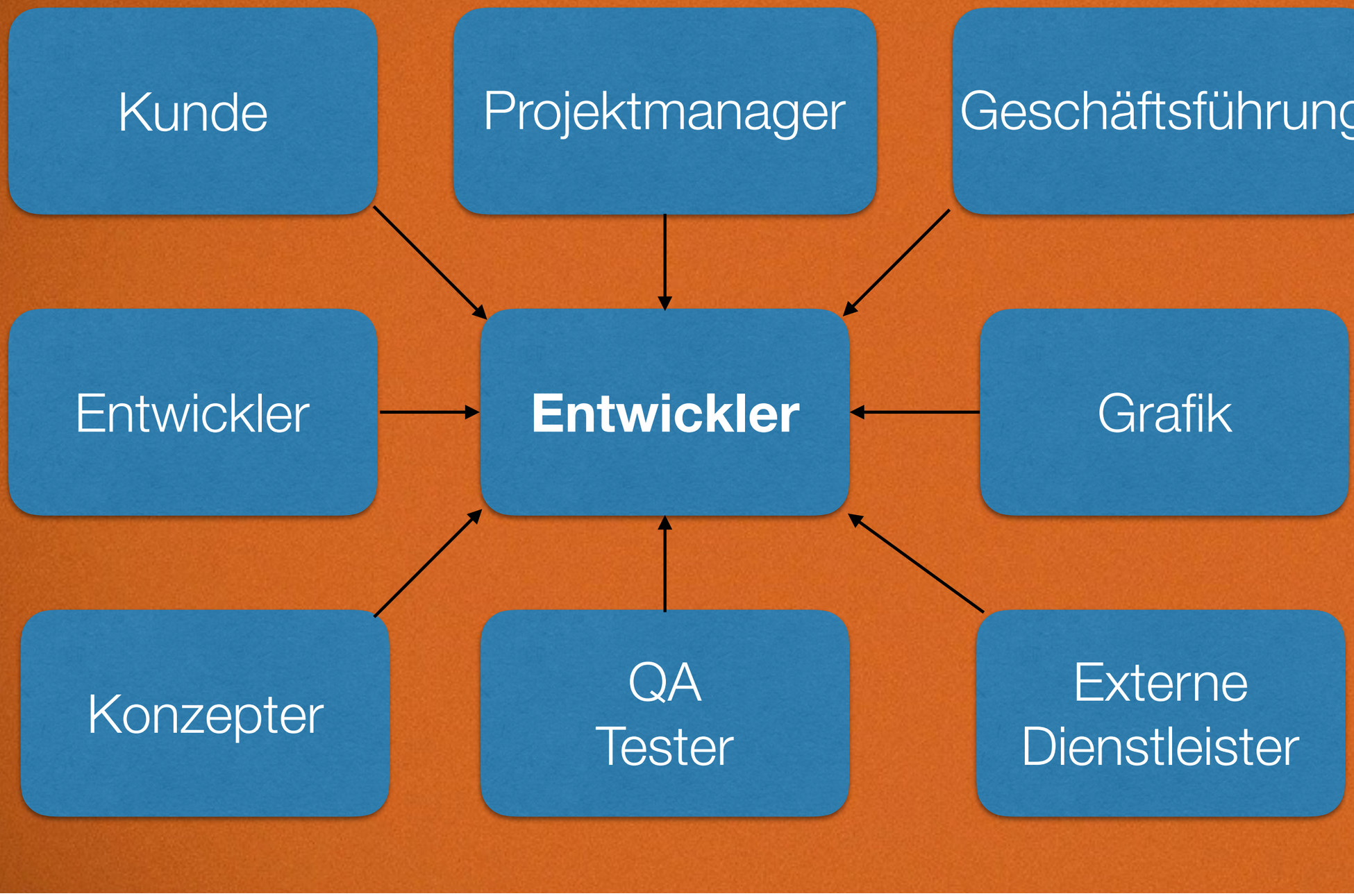
„Der Code ist zum wegschmeissen.“

„Wo sind eure Tests?“

„Das hat die Grafik geschätzt.“

„Hab ihr noch nie von Dependency Injection gehört?“

# „Spannungsfeld“



# Arbeite ich professionell?

# Bin ich ein guter Entwickler?

# Motivation: „Das sollte jeder lesen!“

# „BS-Buchclub“

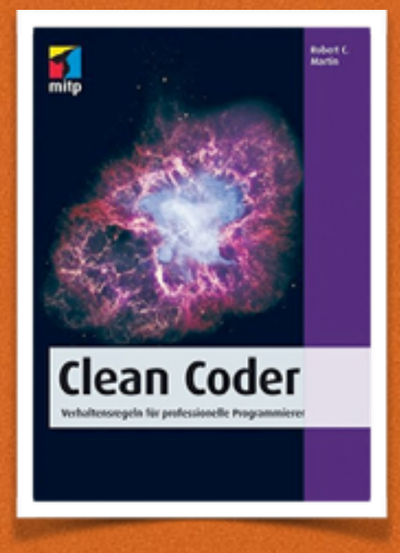
Quelle: Wikipedia

# Robert C. Martin

***„Robert C. Martin, auch bekannt als ‚Uncle Bob', arbeitet seit den 1970er Jahren in diversen Softwareentwicklungsprojekten, seit 1990 als international anerkannter IT-Berater. 2001 initiierte er die Entwicklung des Agilen Manifests, das Fundament agiler Softwareentwicklung. Er ist auch führendes Mitglied der Bewegung Software Craftsmanship die sich der Clean Code Softwareentwicklung verschrieben hat."*** **- Wikipedia**

*"Auf diesen Seiten werde ich versuchen zu dokumentieren, was es heisst, ein professioneller Programmierer zu sein."*

*"Woher weiß ich, was zu diesen Handlungen, Disziplinen und Aktionen gehört? Weil ich sie auf die harte Tour lernen musste."*



# → Übersicht

- **Professionalität**

- **Kommunikation**
  - Nein sagen / Ja sagen / Meetings

- **Programmieren**
  - Bereit sein / Zeitmanagement / Aufwandsschätzungen

- **Testen**
  - Test-Driven-Development / Akzeptanztests / Teststrategien

- **Erkenntnisse**

# Professionalität

# → Professionalität

- **Verantwortung übernehmen!**

- **Richte keinen Schaden an!**

  - Hippokratischer Eid: Tue gutes

- **Beschädige nicht die Funktion!**

  - Keine Bugs produzieren
  - Komplexe Software: Fehlerrate senken
  - Nicht die gleichen Fehler immer wieder machen

# → Professionalität

- **Beschädige nicht die Struktur!**
  - Software soll einfach zu ändern sein
  - In der Lage sein Änderungen vorzunehmen ohne exorbitante Kosten zu verursachen
  - Einfache Änderungen möglich? Wenn nicht anpassen: “Pfadfinder-Regel”
- **Du musst wissen ob es funktioniert!**
  - Teste deinen Code
  - Jede einzelne Code Zeile, die du schreibst, sollte getestet werden
  - Automatisierte QA

# → Professionalität

- **Die QA darf nichts finden!**
  - Nicht wissentlich mangelhaften Code an die QA schicken
  - QA ist kein Bug Netz
  - Unprofessionell: Code freigeben von dem du nicht weiß ob er funktioniert
  - Wenn die QA oder der Benutzer einen Fehler findet, sei verärgert und entschlossen, dass es nächstes mal nicht mehr passiert

# → Professionalität

- **Du solltest dich in deinem Bereich auskennen!**
  - **Design-Pattern**
    - Alle 24 der GoF (Gang of Four - Design Pattern Buch)
  - **Design-Prinzipien**
    - SOLID-Prinzipien
  - **Methoden**
    - XP, Lean, Kanban, Wasserfall, Scrum, …
  - **Disziplinen**
    - TDD, Objektorientiertes Design, Kontinuierliche Integration, …
  - **Artefakte**
    - UML, Strukturierte Diagramme, Petri-Netze, …

# → Professionalität

- **Du solltest dich in deinem Arbeitsgebiet auskennen!**
  - In der Domäne auskennen
  - Programmiere nicht nur nach der Spezifikation, erkenne warum sie für den Geschäftsprozess relevant ist
  - Hinterfrage die Spezifikationen kritisch um Fehler darin erkennen zu können

# → Professionalität

- **Lebenslanges lernen!**
  - Auf dem laufenden halten: Bücher, Artikel, Blogs, Tweets
  - Praxis: Skills außerhalb der normalen Arbeitsanforderungen ausüben
    - Katas / Coding Dojos lösen
- **Mentoring!**
  - Lernen durch Lehren
  - Fühle dich für die Betreuung von Jüngeren persönlich verantwortlich

# Kommunikation

Nein sagen / Ja sagen / Meetings

# → Kommunikation: Nein sagen

- **„Der Entwickler als Superheld“**
  - “*Profis sagen angesichts der Macht die **Wahrheit**. Profis haben den Mumm, ihren Managern gegenüber Nein zu sagen.*”
  - *“Sklaven dürfen nichts ablehnen. Arbeiter könnten dabei zögern, etwas zu verweigern. Aber von Profis erwartet man, dass sie Nein sagen.“*

# → Kommunikation: Nein sagen

- **Feindliche Rollen**

  - **Manager** verfolgen ihren Job und verteidigen ihre Ziele: „Das muss bis morgen fertig sein!“

  - Es ist nicht zu schaffen aber du sagst: „Okay ich probiere es“

  - **Verteidige ebenfalls deine Ziele!**

  - Zum bestmöglichen Ergebnis kommen: **Ziel welches du und dein Manager gemeinsam teilen!**

  - Ziel finden durch **Verhandeln**

  - Die Frage nach dem **Warum**: Details zum Verständnis liefern

# → Kommunikation: Nein sagen

- **Versuchen**
  - Es gibt kein Versuchen
  - Versuchen: „Besondere Mühe aufwenden“
  - Ziel durch Erbringung besonderer Leistung erreichen
  - Wenn du es nicht schaffst, hast du versagt

- **Teamplayer sein**
  - Regelmäßig kommunizieren und Teamkollegen im Auge behalten
  - Dinge in deinem Verantwortungsbereich so gut wie möglich umsetzen

- **Passive Aggression**
  - Beispiel: PM berichtet nicht über Kalkulation mit GF
  - PM ins Messer laufen lassen: stillschweigende Aggression
  - Nicht stillschweigend hinnehmen sondern **warnen!**

# ➔ Kommunikation: Nein sagen

- **Die Kosten eines Ja**

  - Weg um zum richtigen Ja zu kommen, keine Angst vor einem Nein haben

  - Unmögliche Deadline akzeptieren, weitere Features akzeptieren, du bist in der Verantwortung!

  - „*Profis sind Helden, aber nicht weil sie es darauf anlegen, Profis werden zu Helden wenn sie einen Auftrag gut, rechtzeitig und im budgetierten Rahmen erledigen.*“

# ➔ Kommunikation: Ja sagen

- **Verbindliche Sprache**
  - Sagen, Meinen, Machen
  - Commitment:
    - Du ***sagst***, das du es machst
    - Du ***meinst*** es
    - Du ***machst*** es dann auch tatsächlich
  - „Ich werde bis …“

- **Mangelnde Selbstverpflichtung**
  - „Dazu komme ich wahrscheinlich bis zum Feierabend!“
  - Sollte / müsste
  - Hoffe / Wünsche
  - Wir können …

# → Kommunikation: Ja sagen

- **„Es funktioniert nicht, weil ich mich dafür auf Person X verlassen muss“**
  - Meeting mit Kollegen ansetzen um Schnittstellen zu definieren und Abhängigkeiten zu klären

- **„Es wird nicht funktionieren, weil ich nicht genau weiß, ob man es machen kann“**
  - Commitment: Herausfinden ob es umsetzbar ist
  - Commitments die dich dem Ziel näher bringen

- **„Es wird nicht funktionieren, weil ich es manchmal einfach nicht schaffe“**
  - Alarm schlagen: Team innehalten, Situation neu beurteilen und Aktionen treffen
  - Commitment anpassen

# ➔ Kommunikation: Ja sagen

- **Lernen wie man Ja sagt**

  - Eigene **Ungewissheit** beschreiben: „Möglich, kann aber auch Montag werden.“

  - **Standards** bewahren: Code muss getestet werden, Code muss sauber sein

  - Nicht zu allem Ja sagen

  - **Kreative Wege** finden um sein Ja möglich zu machen

# → Kommunikation

- **Meetings**
  - Meetings sind **notwendig**
  - Meetings sind **Riesenzeitverschwendung**

  - Leiste aktiven Widerstand gegen Meetings die dir keinen direkten oder wesentlichen Vorteil bringen

  - Meeting nur akzeptieren wenn es für die Arbeit, die du gerade ausführst, direkt erforderlich ist

  - Deine Anwesenheit ist nicht mehr erforderlich: Verabschiede dich aus dem Meeting

  - Meetings mit Agenda: Ziel des Meetings muss klar sein

# Programmieren

Bereit sein / Zeitmanagement / Aufwandsschätzungen

# → Programmieren

**Bereit sein! - nicht müde oder abgelenkt!**

- Ihr Code muss funktionieren. Sie müssen verstanden haben welches Problem sie lösen wollen

- Ihr Code muss das (wahre) Problem des Kunden lösen

- Ihr Code muss gut ins existierende System passen

- Ihr Code muss von anderen Programmierern lesbar sein

# → Programmieren

- **Flow Zustand**
  - **Vermeide die Zone**: Größere Zusammenhänge aus dem Blick verlieren
  - Zone: **Unkommunikativer Zustand**
  - **Paarprogrammierung**: Intensive und konstante Kommunikation

- **Unterbrechungen**
  - Es wird Unterbrechungen geben die dich ablenken und Zeit kosten. Aber auch du wirst Hilfe durch unterbrechen anderer in Anspruch nehmen.
  - Höfliche Bereitschaft hilfsbereit zu sein
  - **Paarprogrammierung**: Sie am Telefon, Partner behält Kontext offen
  - **Test-Driven-Developmen**t: Test bewahrt Kontext

# → Programmieren

- **Sorgencode**
  - Finger vom Code lassen
  - Zeitrahmen setzen in dem sie sich mit der Sorge beschäftigen

- **Schreibblockaden**
  - Paarprogrammierung
  - **Kreativer Input**: Lesen (Software, Politik, Physik, Chemie, Thriller, …)

# → Programmieren

- **Hilfe**
  - „***Programmieren ist tatsächlich dermaßen schwer***, *dass es die Fähigkeiten einer Person übersteigt, das gut zu machen. Egal wie geschickt und bewandert sie sind, sie werden mit Sicherheit von den Gedanken und Ideen eines anderen Programmierers profitieren.*“

- **Anbieten:**
  - Verantwortung sich gegenseitig zur Verfügung zu stehen

- **Annehmen:**
  - Akzeptiere Hilfsangebote
  - Versuche nicht dein Revier zu verteidigen
  - Melde deinen Hilfebedarf an. Lehne dich nicht zurück und akzeptiere dein Feststecken!

# → Programmieren

- **In Verzug sein**
  - Frühzeitige Erkenntnis: Optimal, Normalfall, Schlimmstenfalls
  - Hoffnung ist der Killer: Notfallplan
  - Bleibe bei deiner Kalkulation: Reduziere den Umfang um den Zeitplan einzuhalten
  - Abkürzungen nehmen -> Tests weglassen -> Katastrophe!

- **Überstunden**
  - Zeitweise möglicherweise hilfreich
  - Gehen nach hintern los wenn mehr als zwei Wochen anhaltend
  - Kurzer Zeitraum und Notfallplan

- **Unlautere Lieferung**
  - Behaupten man sei fertig, auch wenn das gar nicht stimmt
  - Maßnahmen: Definiere „Fertig und erledigt“, Akzeptanztests

# → Programmieren

- **Aufwandsschätzungen**
  - „*Einfach und doch furchteinflößend*“
  - „*Der Keil zwischen Business und Entwickler*“

  - **Business**: Sehen sie als Commitment
  - **Entwickler**: Sehen sie als Vermutung oder Mutmaßung

  - **Commitment**:
    - Muss man umsetzen und erreichen. Du weisst, dass du es schaffst
    - Gewissheit: Andere bauen auf deinen Commitments auf

  - **Aufwandsschätzung**:
    - Ist eine Mutmaßung. Nicht unehrenhaft sie nicht einzuhalten
    - **Wir nehmen Aufwandsschätzungen weil wir nicht wissen wie lange es dauert**

# → Programmieren

- **Aufwandsschätzungen**
  - Ist kein konkreter Wert, sondern eine statistische Verteilung: Wahrscheinlichkeitsverteilung

  - **PERT**
  - „Program Evauluation and Review Technique“
  - Art und Weise wie Aufwandsschätzungen berechnet werden

  - **O**: Optimale Schätzung
  - **N**: Standard Schätzung
  - **P**: Pessimistische Schätzung

# Testen

Test-Driven-Development / Akzeptanztests / Teststrategien

# → Testen

- **Test-Driven-Development**
  - „*Wie kannst du wissen, dass dein gesamter Code funktioniert?*“

  - Schreibe erst Produktivcode wenn du vorher einen gescheiterten Unit-Test geschrieben hast
  - Schreibe nicht mehr Produktivcode als nötig ist um den aktuell misslingenden Unit-Test zu bestehen

  - Vorteile:
    - Gewissheit: Viele Tests und Testabdeckung
    - Schlechten Code fixen wenn du ihn siehst
    - Dokumentation: Unit-Tests beschreiben das Design des Systems auf der untersten Ebene

  - TDD ist keine Religion noch magische Zauberformel

# → Testen

- **Akzeptanztests**
  - Anforderungen kommunizieren
  - Von Business, QA und Programmierern geschrieben
  - Definieren ob Anforderungen erfüllt sind

- **Klarheit und Präzision über Systemverhalten**
- Bewahren davor das falsche System zu implementieren und erlauben dir zu wissen wann du **wirklich** fertig bist

- **Automatisierte Tests** (Kosten)
- Wenn von Entwicklern geschrieben, dann anderer Entwickler als der der das System implementiert hat

- **Fazit**: Einziger Weg um Kommunikationsfehler zwischen Business und Entwicklern zu minimieren besteht in autom. Akzeptanztests.

# → Testen

- **Teststrategien**
  - Unit und Akzeptanztests reichen nicht aus
  - Pyramide der Testautomatisierung:
    - Unit-Tests
    - Komponententests
    - Integrationstests
    - Systemtests
    - Manuelle explorative Tests

- Die QA gehört zum Team
- Zusammenarbeit mit der QA

- **Ziel**: Maximales Feedback und Sicherstellung, dass das System sauber bleibt

# Erkenntnisse

# → Erkenntnisse

- **Zweier Teams** (Junior und Prof./Senior)
  - Ein Vollzeit Verantwortlicher Entwickler pro Projekt
- **Aufwandsschätzung nach PERT** (Estimate-Tool)
- **Kommunikation**
  - Niemals versuchen: Verbindliche Aussagen treffen
  - Einheitliche Sprache sprechen
  - Unrealistische Forderungen seitens Projektmanager hinterfragen
- **Business Prozesse kennen**
- **Meetings hinterfragen**
- **Design-Patterns und SOLID-Prinzipien kennen**
  - Vorträge
- **Technische Qualitätssicherungsmaßnahmen treffen**
  - Akzeptanztests in allen Projekten einziehen
  - Pull-Requests nur in Ausnahme ohne Tests

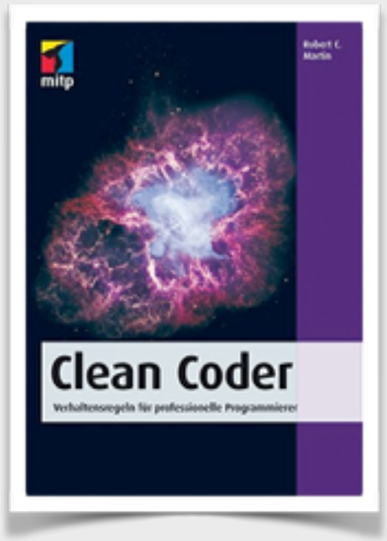
# → Literatur und Quellen

- **Clean Coder** (Robert C. Martin - mitp)

- **Clean Code** (Robert C. Martin - mitp)

- **Agile Software Development** (Robert C.Martin - Prentice Hall)

- **Design Patterns** (Gamma, Helm, Johnson, Vlissides - mitp)






# Vielen Dank für eure Aufmerksamkeit!

Fragen?