# Design Pattern

…und ein bisschen Clean Code

# Euch erwartet…

… Slides mit **Design Pattern** und Clean Code
… **Gruppenarbeit** mit Aufgaben zu den Themen
… Brownfield Projekt auf *github* für das **Refactoring**
… **Lösungen** auf *github* zum weiterarbeiten

# Wer isT dieser Thomas eigentlich

**Vita**

Thomas Ley
45 Jahre
Patchwork

**Hobbys**

Mountainbiken, Sport
Fotografie, Judo
ESP32

**Getting Started**

1994 mit C64
C# seit 2005

**Projekt**

Architekt
DevOps
Coach
Refactoring

**Geht durch's Ohr**

Aus gegebenem Anlass:

Karnevalsmusik ☺

**Find me**

CodeQualityCoach.de

LinkedIn

# Aufgabe

## Wer bist du?

Name

Erster PC/Programmiersprache

Erfahrung mit C# und Design Pattern

Lieblingsmusiktitel

# Zeitplan

12:00

**Start**

**Pause**

13:15 & 14:45

16:00

**Ende**

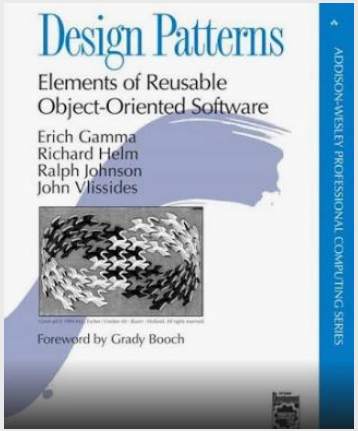
# Disclaimer

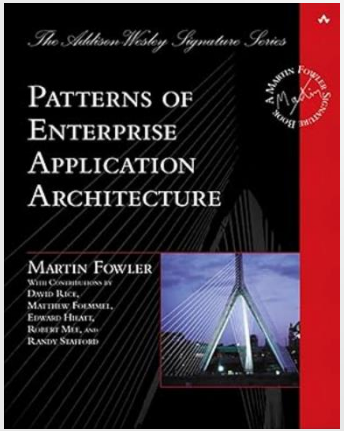
# Design Pattern ∞ Clean Code Principles

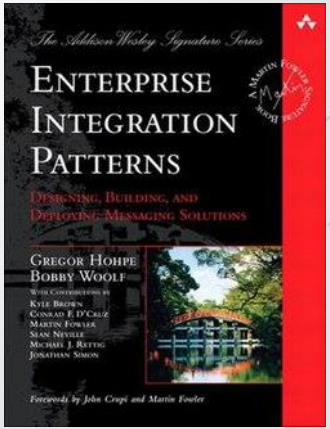
- Design Pattern
    a. Problem → Lösung
    b. Kommunikation
- Clean Code Principles
    a. Strukturierung von Code
    b. Qualität von Code
    c. Dokumentation
- “Das Ganze ist mehr als die Summe seiner Teile”

# Design Pattern

- Design Pattern – Gamma, Helm, Johnson, Vlissides (Gang of Four, GoF)
- Patterns of Enterprise Application Architecture – Fowler, et.al.
- Enterprise Integration Patterns – Hohpe, Woolf, et.al.

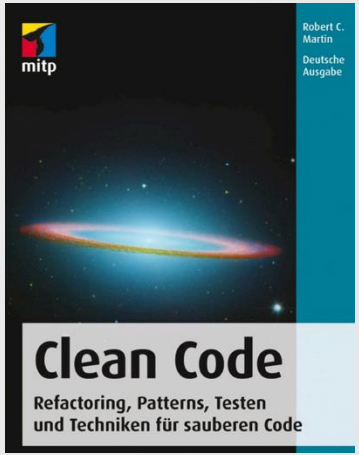


# Clean Code

- Buch: Clean Code – Martin
- Buch: The Pragmatic Programmer – Hunt, Thomas
- Link: https://clean-code-developer.de/




# SOLID Principles

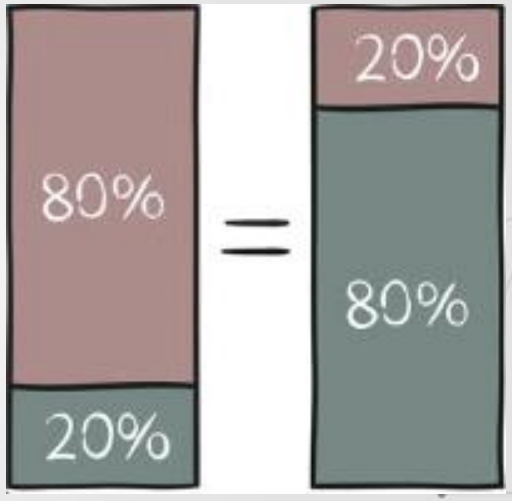
- Abkürzung aus Abkürzungen, also sehr wichtig ☺

- SRP (Single Responsibility Principle)
- OCP (Open Closed Principle)
- LSP (Liskov Substitution Principle)
- ISP (Interface Seggregation Principle)
- DIP (Dependency Inversion Principle)

# Pareto Prinzip

- Auch Pareto-Effekt oder 80-zu-20-Regel
- 80% der Arbeit in 20% der Zeit

- 100% → Over Engineering?
- 100% → Lost in Details?
- 100% → Quatschen, nicht machen?



# Think Big, Start Small

- Was muss ich jetzt berücksichtigen?
- Was kann ich später implementieren?
- Was kann ich (theoretisch) austauschen?
- Wie aufwändig ist eine (Ver-)Änderung?

# Schnittstellen & Abhaengigkeiten

Es war einmal...

# 01

# Interfaces

Er ist ein Mensch, er ist ein Tier, er ist …

# Interfaces

- Klasse ohne Implementierung (“abstract class/method”)
- Definition der Funktionen → Signaturen
- Vertrag über die implementierte Funktionalität
- Unabhängig von der konkreten Implementierung
- Polymorphie (Vielgestaltigkeit) ohne “Diamond of Death”
    a. Eine Klasse, mehrere Interfaces
    b. Ein Interface, mehrere Implementierungen

# Interface Segregation Principle (ISP)


- Große Schnittstellen in mehrere kleine Schnittstellen aufteilen
- Trennung der Zuständigkeiten
- Zusammenhalten der Abhängigkeiten

# Das github Projekt …

… könnt ihr entweder als *zip* herunterladen

… oder mittels *git clone* auschecken,
und auf einem *Branch-per-Aufgabe* arbeiten.

# Aufgabe

## Person “interfacen”

Projekt: “Person”

Aufgabe: Sinnvolle *Interfaces* extrahieren

# David Wheeler

- “All problems in computer science can be solved by another level of indirection.”
- Definieren von Abstraktionen für die Kommunikation

- Schnittstellen als gemeinsamer Vertrag
  - Interface
  - Typen
  - Methodensignaturen

➔ Alles was “public” ist, ist der Vertrag mit meiner API

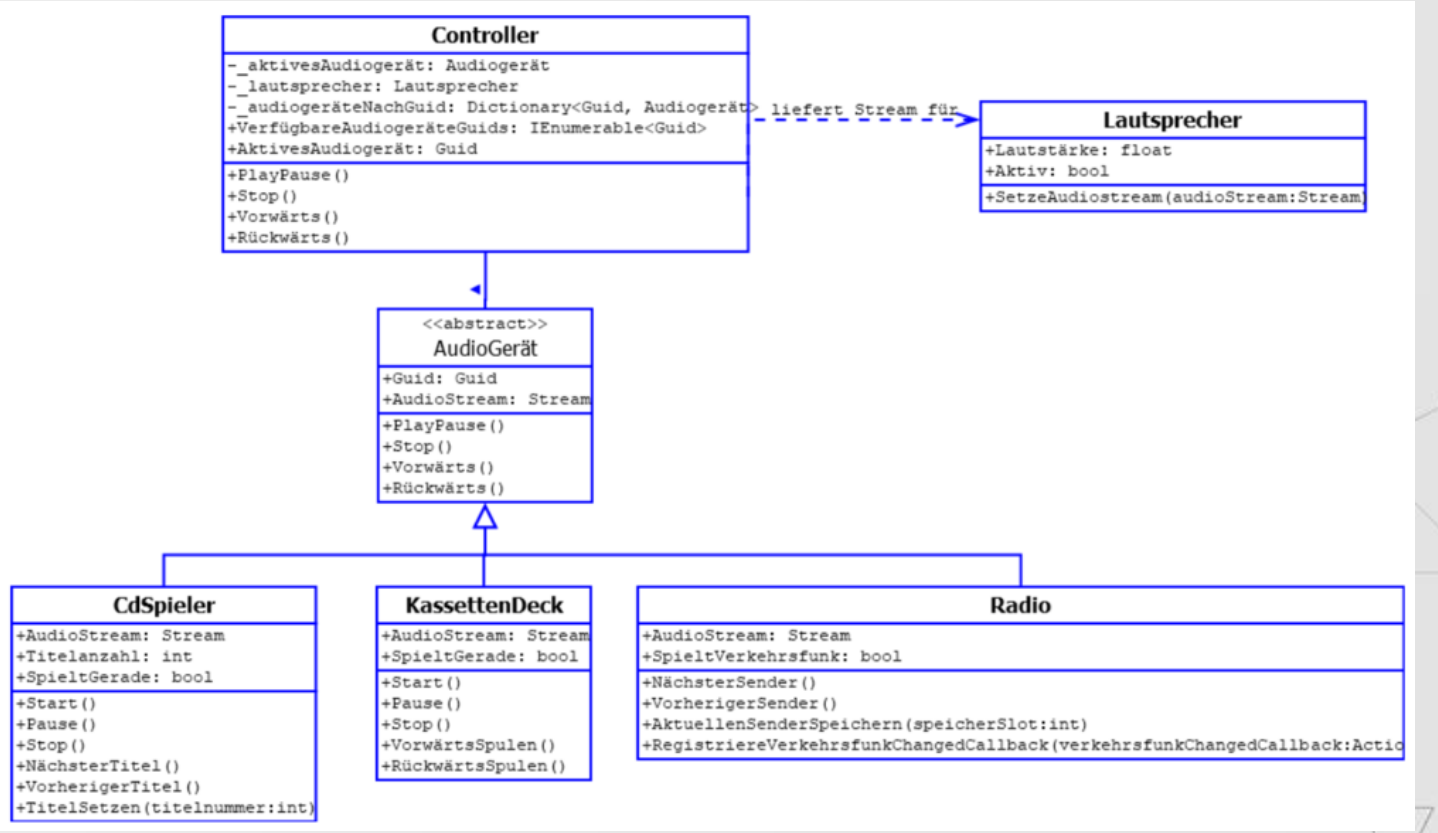
# Liskov Substitution Principle (LSP)


- Das LSP besagt, dass Subtypen sich so verhalten müssen wie ihr Basistyp

- Löst der Basistyp keine Exception aus, dürfen Subtypen auch keine Exception werfen
- Der Subtyp darf die Funktionalität eines Basistyps lediglich erweitern (z.B. Wertebereich)

→ Besser “Favour Composition over Inheritance”

# Ist das LSP?


Controller
-_aktivesAudiogerät: Audiogerät
-_lautsprecher: Lautsprecher
-_audiogeräteNachGuid: Dictionary<Guid, Audiogerät>
+VerfügbareAudiogeräteGuids: IEnumerable<Guid>
+AktivesAudiogerät: Guid
+PlayPause()
+Stop()
+Vorwärts()
+Rückwärts()
liefert Stream für
Lautsprecher
+Lautstärke: float
+Aktiv: bool
+SetzeAudiostream(audioStream:Stream
<<abstract>>
AudioGerät
+Guid: Guid
+AudioStream: Stream
+PlayPause()
+Stop()
+Vorwärts()
+Rückwärts()
CdSpieler
+AudioStream: Stream
+Titelanzahl: int
+SpieltGerade: bool
+Start()
+Pause()
+Stop()
+NächsterTitel()
+VorherigerTitel()
+TitelSetzen(titelnummer:int)
KassettenDeck
+AudioStream: Stream
+SpieltGerade: bool
+Start()
+Pause()
+Stop()
+VorwärtsSpulen()
+RückwärtsSpulen()
Radio
+AudioStream: Stream
+SpieltVerkehrsfunk: bool
+NächsterSender()
+VorherigerSender()
+AktuellenSenderSpeichern(speicherSlot:int)
+RegistriereVerkehrsfunkChangedCallback(verkehrsfunkChangedCallback:Actio

# 02

# Abhaengigkeiten

# Frage

## Abhaengigkeiten

Gibt es “unabhängigen” Code?

# Abhaengigkeiten - “New is Glue”

- New() kennt die Implementierung
- New() kennt den Lebenszyklus (Lifecycle)
- New() kennt die Konstruktor Anhängigkeiten

→ Creational Pattern

# Dependency Inversion Principle (DIP)


- “dependency inversion principle is a specific methodology for loosely coupling software modules”

- Module höherer Ebenen sollten nicht von Modulen niedrigerer Ebenen abhängen. Beide sollten von Abstraktionen abhängen.

- Abstraktionen sollten nicht von Details (Implementierungen) abhängen. Details (Implementierungen) sollten von Abstraktionen abhängen.

Quelle Wikipedia

# Abhaengigkeit umkehren

- Anhängigkeiten werden über den Konstruktor angefordert
- Modul kennt nur noch die Abstraktion (Interface)
- “new()” verschwindet aus der Klasse

→ DI Frameworks in größeren Applikationen

# Behavioural Patterns

# 01

# Strategy Pattern


# Single Responsibility Principle

- Eine Funktionseinheit (Methode/Klasse/…) implementiert nur einen Aspekt
- Änderungen unabhängig von anderen Funktionseinheiten

- Anti-Pattern
  - Ravioli Code (Viele sehr kleine Klassen)
  - Functor Object (Klassen mit genau einer Methode)

# Strategy Pattern

- Strategie Entwurfsmuster
- Mehrere Implementierungen einer (ähnlichen) Funktionalität
- Typischerweise Abstrahierung von (technischer) Infrastruktur
- “Extract Interface” für Testbarkeit

- Code Smell: “if, ifelse, ifelse, ifelse, else”

# Aufgabe

## Fertigstellen/Implementieren des Strategy Pattern

Projekt „PdfTools“

IStrategy Interface
Do() Methode

# 02

# Command Pattern

# Command Pattern

- Kommando Entwurfsmuster
- Verwendung z.B. Buttons einer UI

- Prüfen ob ein Kommando ausgeführt werden kann (CanExecute)
- Ausführen des Kommandos(Execute)
- Beide Methoden bekommen den gleichen Parameter “Context”

- Code Smell: “if, ifelse, ifelse, ifelse, else”

# Aufgabe

## Implementieren des Command Pattern

Projekt „PdfTools“

ICommand Interface
CanExecute(ctx)
Execute(ctx)

# 03

# “Simple” Composite Pattern

# “Simple” Composite Pattern

- Aggregieren von mehreren Implementierungen
- Aufrufen als wäre es eine Implementierung
- CompositeXYZ erbt von IXYZ
- CompositeXYZ kennt alle “regulären” XYZ
- CompositeXYZ leitet alle Aufrufe an XYZ weiter

# Aufgabe

## Implementiere das Composite Command

Projekt “PdfTools”

CompositeCommand erstellen
Commands und injizieren

Tagging Interface verwenden

# 04

# Empty-Object Pattern

aka. Null-Object Pattern

# Null-Object Pattern / Empty-Object Pattern

- Implementieren des Interface mit einer „leeren“ Funktionalität
- Funktionen geben „Default()“ zurück
- Methoden machen „nichts“

- Das Empty-Object Pattern darf den Zustand/das Verhalten des Systems nicht verändern

# Aufgabe

## Implementiere ein Empty-Object

Projekt “PdfTools”

EmptyStrategy,
Fallback-Strategy.

# Resumee Command vs. Strategy

- Prüfen „CanExecute()“
- Mehrere „true“ für gleichen Context
- Verwendung: z.B. UI-Buttons

- N-Methoden per Strategie
- I.d.R. nur eine Strategie
- Verwendung: z.B. MSSQL vs. MongoDB

- Aggregierung über “Simple” Composite

- Leere Implementierung Empty-Object