DHBW
Duale Hochschule
Baden-Württemberg
Karlsruhe

# Programmentwurf

## Dart Counter

im Rahmen der Prüfung zum

**Bachelor of Science (B.Sc.)**

des Studienganges Informatik

an der Dualen Hochschule Baden-Württemberg Karlsruhe

von

**Robin Purschwitz**

| | |
|---|---|
| Abgabedatum: | 25. Mai 2023 |
| Bearbeitungszeitraum: | 12.12.2022 - 25.05.2023 |
| Matrikelnummer, Kurs: | 2415691, TINF20B2 |
| Gutachter der Dualen Hochschule: | Dr. Lars Briem |

# Inhaltsverzeichnis

# Abkürzungsverzeichnis

**SRP** Single Responsibility Principle

**OCP** Open-Closed Principle

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Quellcodeverzeichnis

# 1 Einführung

## 1.1 Übersicht über die Applikation

## 1.2 Wie startet man die Applikation

## 1.3 Wie testet man die Applikation

# 2 Clean Architecture

In diesem Kapitel steht die Clean Architecture und deren zentrale Aspekte im Fokus. Zuerst erfolgt eine Analyse der Dependency Rule, einer Schlüsselregel der Clean Architecture. Untersucht werden dabei die Auswirkungen dieser Regel auf die Softwarearchitektur, ergänzt durch positive und negative Anwendungsbeispiele.

Im Anschluss daran richtet sich der Fokus auf die Struktur der Clean Architecture, wobei die einzelnen Schichten detailliert analysiert werden. Besondere Aufmerksamkeit gilt dabei den Schichten "Domainünd Äpplication", deren Rolle und Bedeutung innerhalb der Clean Architecture ausführlich diskutiert werden. Diese Analysen ermöglichen ein tiefgreifendes Verständnis der Clean Architecture und ihrer praktischen Anwendung.

## 2.1 Was ist Clean Architecture

Die *Clean Architecture*, auch bekannt als die *Onion Architecture*, ist ein Software-Entwurfsprinzip, das von Robert C. Martin, entwickelt wurde. Sie zielt darauf ab, eine klare und getrennte Struktur in Software-Systemen zu schaffen, um Wartbarkeit, Testbarkeit und Flexibilität zu verbessern.

Die Clean Architecture teilt eine Anwendung in konzentrische Schichten auf, wobei jede Schicht bestimmte Arten von Aufgaben erfüllt und klar definierte Abhängigkeiten aufweist. Die Schichten, von innen nach außen, sind in der Regel wie folgt:

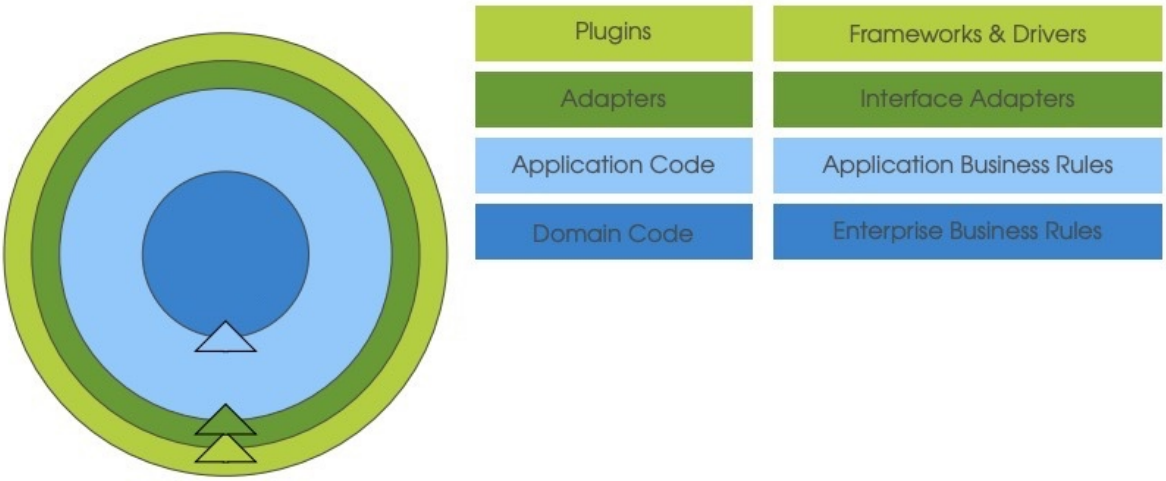


Abbildung 2.1: Clean Architecture Schichten

**Domain-Schicht**: Dies ist die innere Schicht, die die Geschäftslogik und die Geschäftsregeln einer Anwendung enthält. Sie hat keine Abhängigkeiten von den äußeren Schichten und repräsentiert die fundamentalen Konzepte der Anwendung, unabhängig von spezifischen technologischen Details.

**Anwendungs-Schicht**: Diese Schicht enthält spezifische Geschäftslogik, die sich auf bestimmte Anwendungsfälle bezieht. Sie ist von der Domain-Schicht abhängig und kann mit ihr interagieren, aber sie kennt keine Details über äußere Schichten.

**Adapter-Schicht**: Diese Schicht übersetzt Daten zwischen den Formaten, die für die inneren Schichten und die äußeren Schichten geeignet sind. Sie könnte beispielsweise Datenbankcode, Benutzeroberflächen-Code oder sogar Code für externe Dienste enthalten.

**Plugin-Schicht**: Dies ist die äußerste Schicht, die spezifische Technologien wie Datenbanken, Webserver oder Frameworks umfasst. Sie interagiert mit den inneren Schichten durch Ports und Adapter.

Das Hauptprinzip der Clean Architecture ist die Regel der Abhängigkeitsrichtung: Abhängigkeiten sollten immer von äußeren Schichten zu inneren Schichten gerichtet sein. Dies bedeutet, dass der Code in den inneren Schichten unabhängig von spezifischen Frameworks, Datenbanken oder anderen Technologien ist, was ihn einfacher zu testen und zu warten macht.

Zusätzlich wird durch die klare Trennung der Verantwortlichkeiten die Einhaltung des Single Responsibility Principle (SRP) und des Open-Closed Principle (OCP) aus den SOLID-Prinzipien erleichtert. Es ermöglicht auch eine bessere Modularität und Austauschbarkeit der Komponenten, da Änderungen in einer Schicht sich nicht auf die anderen Schichten auswirken sollten.

Zusammenfassend lässt sich sagen, dass die Clean Architecture ein Ansatz ist, der darauf abzielt, die Unordnung und Komplexität in Softwareprojekten zu reduzieren, indem klare Grenzen und Regeln für die Struktur und Organisation des Codes vorgegeben werden. Sie ermöglicht es Entwicklern, Systeme zu erstellen, die widerstandsfähig gegenüber technologischen Änderungen sind und die sich im Laufe der Zeit leicht anpassen und erweitern lassen.

## 2.2 Analyse der Dependency Rule

## 2.3 Positiv-Beispiel: Dependency Rule

## 2.4 Negativ-Beispiel: Dependency Rule

## 2.5 Analyse der Schichten

### 2.5.1 Schicht: Domain

### 2.5.2 Schicht: Application

# 3 SOLID

## 3.1 Analyse Single-Responsibility-Principle (SRP)

### 3.1.1 Positiv Beispiel

### 3.1.2 Negativ Beispiel

## 3.2 Open Closed Principle (OCP)

### 3.2.1 Positiv Beispiel

### 3.2.2 Negativ Beispiel

## 3.3 Analyse Liskov-Substitution- (LSP), Interface-Segregation- (ISP), Dependency-Inversion-Principle (DIP)

### 3.3.1 Positiv Beispiel

### 3.3.2 Negativ Beispiel

# 4 Weitere Prinzipien

## 4.1 Analyse GRASP: Geringe Kopplung

### 4.1.1 Positiv Beispiel

### 4.1.2 Negativ Beispiel

## 4.2 Analyse GRASP: Hohe Kohäsion

## 4.3 Don't Repeat Yourself (DRY)

# 5 Unit Tests

## 5.1 10 Unit Tests

## 5.2 ATRIP: Automatic

## 5.3 ATRIP: Thorough

## 5.4 ATRIP: Professional

## 5.5 Code Coverage

## 5.6 Fakes und Mocks

# 6 Domain Driven Design

## 6.1 Ubiquitous

## 6.2 Entities

## 6.3 Value Objects

## 6.4 Repositories

## 6.5 Aggregates

# 7 Refactoring

## 7.1 Code Smells

## 7.2 2 Refactorings

# 8 Entwurfsmuster

## 8.1 Entwurfsmuster:

## 8.2 Entwurfsmuster: