

# Prinzipien und Anwendung des Software Designs anhand von Schichten- und Hexagonaler Architektur

Simon Thalmaier  
Technische Hochschule Ingolstadt  
Informatik  
Matrikelnummer: 00108692

**Zusammenfassung—**  
**Index Terms—Software, Design, Hexagonale Architektur, Schichtenarchitektur, SOLID**

## I. EINLEITUNG

Seit Jahrzehnten haben sich Softwaresysteme und ihre Architektur weiterentwickelt. Von den damals weitverbreiteten monolithischen Anwendungen sehen wir aktuell einen Aufschwung an kleinen, modulierten Microservices. Immer mehr Geräte haben eingebaute Software und die Größe bzw. Komplexität von Sourcecode steigt mit jedem Jahr, dadurch auch die verbundenen Entwicklungskosten. Immer mehr wandert der Fokus bei Softwareentwicklung auf Flexibilität, Übersichtlichkeit und Wartbarkeit. Es haben sich daraus verschiedenste Software Architekturen bewährt.

Im Folgenden wird auf bekannte Design-Prinzipien und zwei konkrete Architekturen eingegangen und analysiert.

## II. PRINZIPIEN DES SOFTWARE DESIGNS

Prinzipien eines idealen Software Designs unterscheiden sich je nach verwendeten Programmierparadigmen und unterliegender Architektur. Diese hängen stark von der Aufgabe und Anforderungen ab, welche das Programm letztendlich erfüllen soll. Um Software zukunftssicher erstellen zu können, wurden über die Jahre Richtlinien definiert, welche Entwickler behilflich sein sollen. Es haben sich hierbei einige Grundsätze durchgesetzt, welche sich auf einzelne Klassen oder auf die Relationen von Module zueinander beziehen. Zunächst werden diese Prinzipien vorgestellt, analysiert und die resultierenden Auswirkungen bei Einhaltung bzw. Nichteinhaltung erläutert.

### A. Das SOLID-Akronym

Von Michael Feathers und Robert C. Martin geprägt, zählen die SOLID Prinzipien zu dem Fundament eines stabilen Designs. Es beschreibt wünschenswerte Eigenschaften von Komponenten und ihre Beziehungen zueinander. Eine Softwarearchitektur sollte somit den Entwickler dabei unterstützen, diese Prinzipien anzuwenden.

1) *Single-Responsibility-Prinzip*: Durch diese Richtlinie soll sichergestellt werden, dass die Verantwortlichkeit eines Moduls zu maximal einem Akteur gehört. Konkret darf jedes Modul nur einmal abgeändert werden müssen, unabhängig davon wie viele Anforderungen sich ändern. Eine allgemeinere Variante besagt, dass jede Variable, Methode, Klasse

usw. genau eine Aufgabe besitzen soll. Bei Verletzung kann eine Anpassung des Codes zu unerwarteten Nebenwirkungen führen. Als Beispiel kann eine Funktion gesehen werden, welche überprüft, ob ein Passwort alle Anforderungen erfüllt. Die gleiche Funktion wird für sowohl Adminaccounts als auch für normale Benutzeraccounts verwendet.

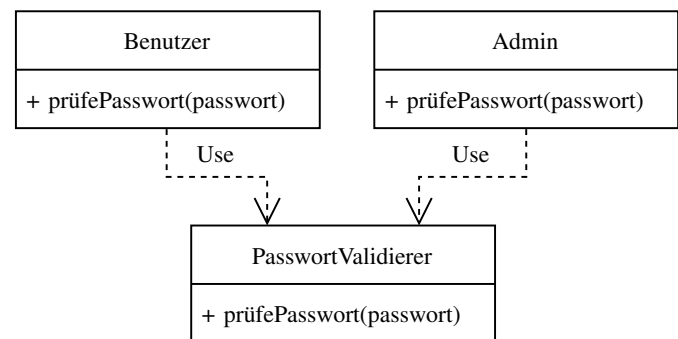


Abbildung 1. Verletzung des Single-Responsibility-Prinzip

Eine neue Anforderung sieht vor, dass Adminaccounts zukünftig eine höhere Mindestlänge besitzt. Eine unaufmerksame Änderung der Funktion hat somit auch eine Auswirkung auf Benutzeraccounts. Dies ist ein Widerspruch des Single-Responsibility-Prinzips. Die Funktionen müssen somit unterschiedliche Implementierungen besitzen.

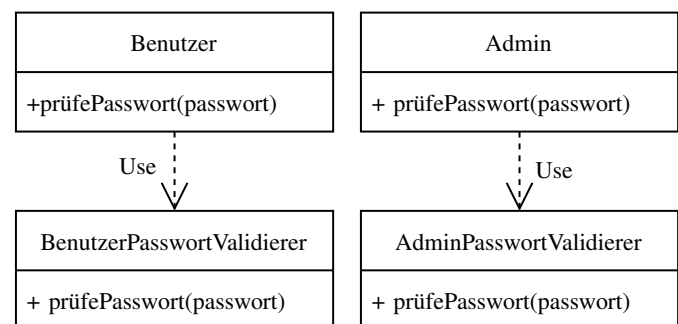


Abbildung 2. Anwendung des Single-Responsibility-Prinzip an dem vorherigen Beispiel

2) *Open-Closed-Prinzip*: Hierdurch werden zwei gewünschte Aspekte eines Moduls beschrieben. Einerseits sollte ein Modul offen sein für Erweiterungen, andererseits geschlossen gegenüber Veränderung. Dies soll es Entwicklern

ermöglichen bereits bestehende Funktionalitäten anzupassen ohne dass der Code, welcher auf diese Funktionsweise basiert, bricht oder abgeändert werden muss. Eine Möglichkeit dieses Prinzip anzuwenden ist das Definieren eines geschlossenen, unveränderbaren Interfaces. Module können diese Schnittstelle und die benötigten Eigenschaften implementieren, aber dennoch jederzeit mit neuen Feldern und Funktionen erweitert werden. Durch Polymorphie ist es weiterhin möglich die konkreten Implementierung hinter dem Interface auszuwechseln. Damit bleiben Abhängigkeiten flexibel und zusätzliche Anforderungen können leichter der Software hinzugefügt werden.

Angewandt an dem Passwortbeispiel ist es möglich ein Interface für den Validierer zu definieren, welches die benötigte Funktion bereitstellt. Obwohl das Interface an sich als geschlossen gilt und somit nicht abgeändert werden sollte, können jederzeit weitere Validierer erstellt werden, welche die bestehende Funktionalitäten implementieren und darüber hinaus erweitern können, wie in Abbildung 3 dargestellt wurde.

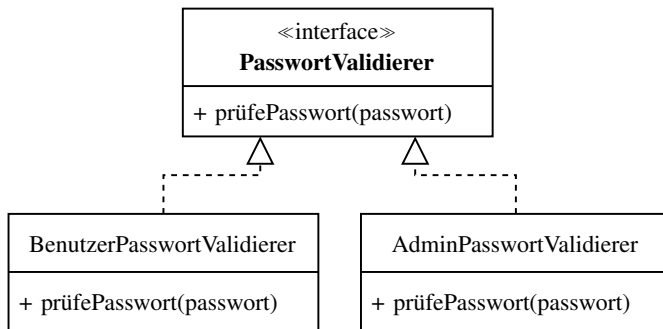


Abbildung 3. Anwendung des Open-Closed-Prinzip

3) *Liskovsches Substitutionsprinzip*: Das in 1994 von Barbara Liskov und Jeannette Wing definierte Prinzip besagt, dass eine Unterklasse S einer Oberklasse T die Korrektheit einer Anwendung nicht beeinflussen soll, wenn ein Objekt vom Typ T durch ein Objekt vom Typ S ersetzt wird. Als Beispiel kann hier die Oberklasse Nutzer und die Unterklasse Admin verwendet werden. Wird nun ein Passwort an eine Funktion zum Speichern von neuen Passwörtern der Nutzer-Klasse übergeben, kann die Methode erfolgreich durchlaufen, obwohl mit dem gleichen Passwort die gleiche Funktion der Admin-Klasse fehlschlägt, da Admin-Passwörter unter strenger Richtlinien liegen. Dies widerspricht dem Substitutionsprinzip.

Um die Einhaltung zu garantieren, sollten die beiden Klassen entweder unabhängig voneinander sein, oder die Passwortspeicherungsfunktion in der Admin-Klasse überschrieben werden.

Das *Liskovsches Substitutionsprinzip* ist von der unterliegenden Architektur unabhängig und bezieht sich auf die Komposition von Klassen und ihre Relationen zueinander. Dadurch beeinflusst eine Architektur diese Richtlinie nicht und kann in den folgenden Analyse vernachlässigt werden.

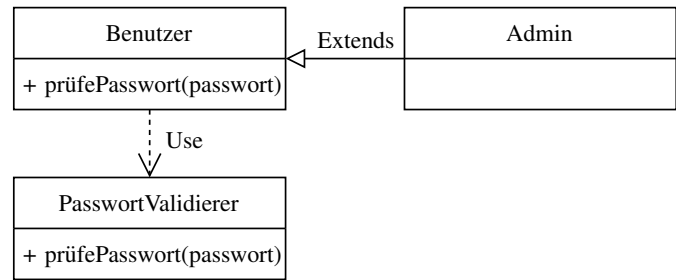


Abbildung 4. Verletzung des Liskovsches Substitutionsprinzip

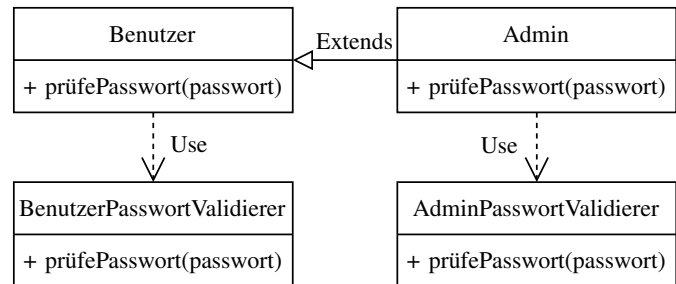


Abbildung 5. Anwendung des Liskovsches Substitutionsprinzip an dem vorherigen Beispiel

4) *Interface-Segregation-Prinzip*: In vor allem monolithischen Systemen finden sich öfters riesige Interfaces mit einer Vielzahl von Funktionen. Das Interface-Segregation-Prinzip besagt, dass Clients nie gezwungen seinen Schnittstellen zu verwenden, welche mehr Funktionalitäten bereitstellen als sie benötigen. Dadurch sollen Interfaces übersichtlich sein und vor allem nach dem *Single-Responsibility-Prinzip* auch nur eine Verantwortung erfüllen.

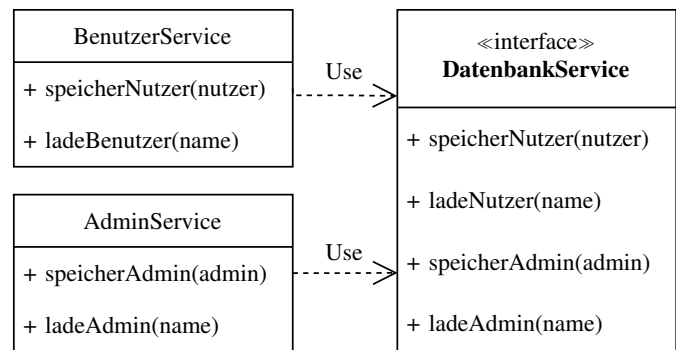


Abbildung 6. Verletzung des Interface-Segregation-Prinzip

5) *Dependency-Inversion-Prinzip*: Um die Software flexibler und anpassbarer zu gestalten, sollten die Module so weit wie möglich unabhängig von anderen Modulen designet werden. Änderungen an den Quelltext bergen das Risiko unerwünschte Nebeneffekte zu erzeugen oder zwingen den Entwickler auch Anpassungen an weiteren Modulen vorzunehmen. Durch eine lose Kopplung sollen solche Situation vermieden werden. Der erste Teil des *Dependency-Inversion-Prinzip* besagt, dass höherliegende Komponenten nicht di-

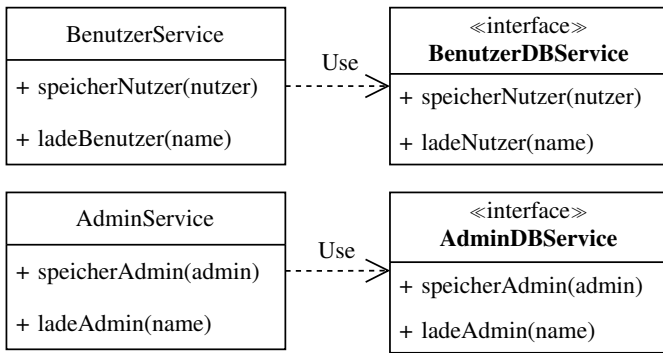


Abbildung 7. Anwendung des Interface-Segregation-Prinzips an dem vorherigen Beispiel

rekt von darunterliegenden Komponenten abhängig sein sollen, sondern die Kommunikation zwischen ihnen über eine abstrakte Schnittstelle geschieht. Der zweite Abschnitt des Prinzips befasst sich wie diese Abstraktion designt wird, um eine höhere Wiederverwendbarkeit der höheren Ebenen zu gewährleisten. Das Interface sollte hiernach nicht an die Implementierung gekoppelt sein, sondern die Details sollten von der Abstraktion abhängen. Bei richtiger Anwendung können dadurch höhere Module, ohne die Korrektheit des Programms zu gefährden, die darunterliegenden Module austauschen solange die Abstraktionsschicht die gleiche ist.

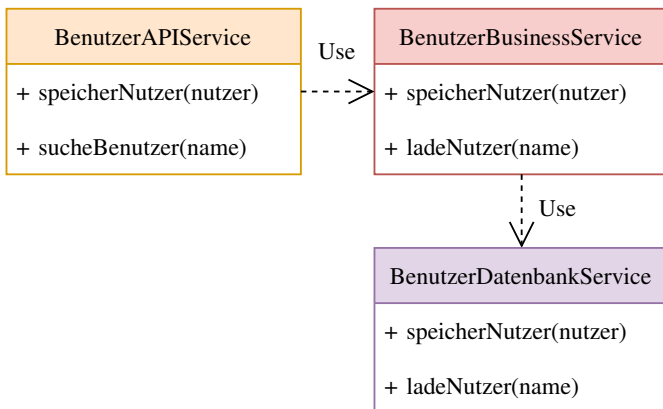


Abbildung 8. Verletzung des Dependency-Inversion-Prinzips

### B. Das GRASP-Akronym

Ausgeschrieben 'General Responsibility Assignment Software Patterns' ist eine Ansammlung von neun Entwurfsmustern, welche in der objektorientierten Programmierung Anwendung finden. Im Kontext von Software Design sind folgende drei von größerer Bedeutung.

1) *Information Expert*: Die Verantwortung zur Lösung eines Domainproblems sollte bei dem Modul liegen, welchem die meisten der benötigten Informationen bereitsteht.

2) *Niedrige Kopplung*: Abhängigkeit zwischen Module bzw. Klassen sollte stets so gering wie möglich gehalten werden, um die Testbarkeit, Wiederverwendbarkeit und zum Schutze von äußeren Änderungen zu steigern.

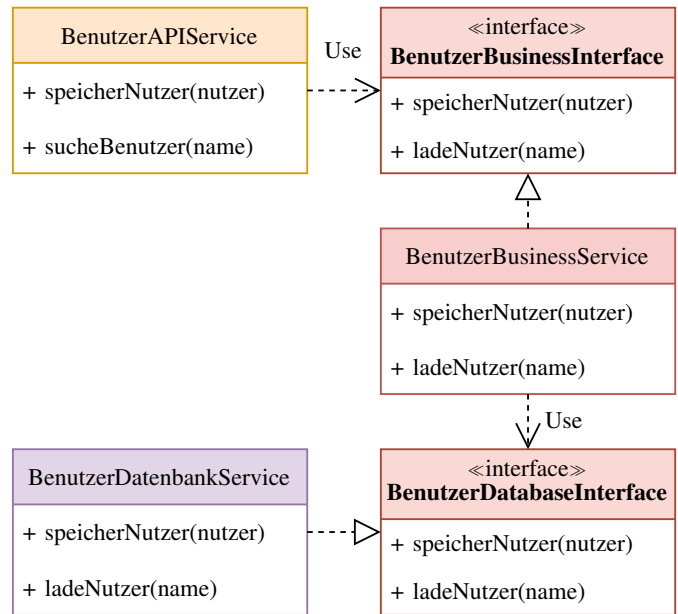


Abbildung 9. Anwendung des Dependency-Inversion-Prinzips anhand einer Schichtenarchitektur

3) *Hohe Kohäsion*: Vergleichbar mit dem *Single-Responsibility-Prinzip* sollten Module eng mit ihrer zugetragenen Aufgabe verbunden sein, wodurch weiterhin eine *niedrige Kopplung* unterstützt wird.

### C. Weitere Messwerte für Architekturdesign

Es gibt zahlreiche weitere Qualitätsattribute, welche je nach Software unterschiedliche Gewichtung tragen. *Testbarkeit* definiert im welchen Maß Module unabhängig voneinander getestet werden können, ohne dabei Änderungen an Architektur oder den Modulen selbst vornehmen zu müssen. *Skalierbarkeit* hingegen beschreibt wie einfach eine Anwendung erhöhte Last abarbeiten kann, indem einzelne Teile bzw. der ganzen Applikation weitere Ressourcen zur Verfügung gestellt werden. Auch im Sinne wie übersichtlich und anpassbar eine Applikation auch bei steigender Codeanzahl bleibt spielt bei Skalierbarkeit eine Rolle. *Einfachheit* charakterisiert im welchen Maß ein Entwickler Erfahrung und Aufwand erfordert die gewählte Architektur umzusetzen.

## III. SCHICHTENARCHITEKTUR

Eine Schichtenarchitektur teilt die Module einer Applikation in verschiedene Ebenen, sogenannte Schichten, ein. Dadurch können Applikationsteile unabhängig voneinander abgeändert oder sogar ganz ersetzt werden. Die Schichtenanzahl variiert je Anwendung, jedoch liegt diese meist zwischen drei und vier. Beispielhaft kann eine Einteilung in Präsentations-, Business- und Datenhaltungsschicht erfolgen. Aufgrund der Eigenschaften einer Schichtenarchitektur wird dieser Ansatz häufig bei simplen CRUD-Applikationen verfolgt. CRUD steht im Kontext der Softwareentwicklung für 'Create Update Delete', somit sind Anwendungen gemeint, welche Daten erzeugen, bearbeiten und löschen mit geringer bis keiner Businesslogik.

Der Kern einer solchen Software sind die Daten selbst, dabei werden Module und die umliegende Architektur angepasst, um die Datenverarbeitung zu vereinfachen.

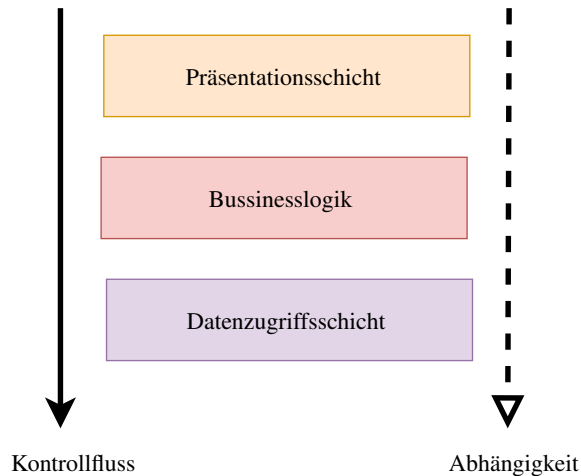


Abbildung 10. Beispielhafte Darstellung einer Drei-Schichtenarchitektur

#### A. SOLID-Prinzipien in der Schichtenarchitektur

Der größte Vorteil dieser Architekturart ist eine erzwungene grobe, natürliche Trennung von Funktionalitäten durch die horizontale Schichteneinteilung. Somit soll verhindert werden, dass eine Komponente beispielsweise Businesslogik und gleichzeitig Zugriff auf die Datenbank regelt. Allerdings ist eine vertikale Trennung nicht gegeben und Module können weiterhin verschiedenen Aufgaben erfüllen. Dadurch ist es möglich, die Schichteneinteilung nicht zu verletzen, jedoch das *Single-Responsibility-Prinzip* zu brechen.

Die klare Aufteilung unterstützt den Entwickler Anforderungen zu definieren, welche eine obere Schicht an darunterliegende Schichten hat. Diese Anforderungen können wiederum als abstrakte Schnittstelle festgelegt werden. Angewandt bedeutet *Dependency-Inversion-Prinzip* auf die Schichtenarchitektur, dass die Details abhängig von diesen Interfaces sein müssen. Beispielsweise gelten Benutzerinterface- und Datenzugriffsschicht als solche Details und sind an die inneren Schichten gebunden. Damit die darüber liegenden Ebenen nicht von Änderungen an den Implementierungen betroffen sind, sollten diese Abstraktionen laut dem *Open-Closed-Prinzip* als geschlossen und die gelten. Ebenso können die Schnittstellen jedoch stetig mit neuen Funktionalität erweitert oder andere konkrete Implementierungen des gleichen Interfaces erstellt werden. Um weiterhin einen SOLID-Ansatz zu verfolgen, müssen wegen dem *Interface-Segregation-Prinzip* die Schnittstellen so klein und präzise wie möglich gehalten werden, damit obere Schichten nur von ihren wirklich benötigten Funktionen abhängig sind.

#### B. Zusätzliche Entwicklungsfaktoren der Schichtenarchitektur

Um eine falsche Analyse bei fehlgeschlagenen Modultest zu vermeiden, müssen Module voneinander isolierbar sein.

Dank der Schichtentrennung können diese unabhängig und isoliert getestet werden, da die Verwendung von Dummy-Objekte, sogenannte Mocks, erforderliche Abhängigkeiten erfüllen können. Bei einer unzureichenden Anwendung der im vorgehenden Absatz analysierten Designprinzipien sind einzelne Applikationsteile gekoppelt, worunter die Testbarkeit stark sinkt und somit die Fehleranfälligkeit steigt. Viele Anwendungen und ihre Anforderungen lassen sich natürlich in verschiedenen Schichten einteilen. Entwickler erhalten mit diesem Architekturstil eine simple, übersichtliche Methode erforderliche Module und ihre Relationen zueinander zu designen. Die Denkweise einer CRUD-Anwendung kann dazu führen, dass eine Software mit einer steigenden Anzahl an Businessanforderungen schlecht skaliert und die Umsetzung der SOLID-Prinzipien vernachlässigt werden, da nicht die Businessregeln im Mittelpunkt stehen sondern die Datenzugriffsschicht, welche allerdings laut dem *Dependency-Inversion-Prinzip* von der Businesslogik abhängig sein sollte. Daraus entsteht möglicherweise eine eng gekoppelte monolithischen Struktur bei der Codeänderungen zu unerwarteten Nebenwirkungen führen kann. Eigenschaften wie Wartbarkeit und Wiederverwendbarkeit steigen mit dem Einhaltungsgrad der SOLID-Prinzipien. Durch die geringe native Unterstützung dieser Prinzipien sind folglich auch die vorher genannten Eigenschaften gefährdet.

### IV. HEXAGONALE ARCHITEKTUR

In der von Alistair Cockburn geprägte Architektur ist der Hauptgedanke die Einteilung von Modulen in Adaptern und Applikationskern. Die Kommunikation zwischen ihnen geschieht hierbei über abstrakte Interfaces, die sogenannten Ports. Daher wird dieser Stil auch *Ports und Adapter Architektur* genannt.

Adapter sind Schnittstellen zwischen externe Systeme und der Businesslogik, wie beispielsweise API-, Datenbank- oder Messaging-Service. Zusätzlich werden sie in zwei Unterkategorien aufgeteilt, die primären und sekundären Adapter, wobei der Steuerungsfluss von den primären Adaptern in die Businesslogik und eventuell weiter über die sekundär Adapter fließt. Ports sind hingegen dem Applikationskern zugeteilt und werden von dieser definiert.

#### A. Architektonischer Vergleich mit einer Schichtenarchitektur

Bei genauer Betrachtung fällt auf, dass Hexagonale Architektur eine umgestellte Schichtenarchitektur ist, welche das *Dependency-Inversion-Prinzip* fest implementiert. Hierbei werden Schichten ohne Businesslogik in den äußeren Ring bewegt und stellen die Adapter dar. Der Applikationskern sind somit die verbleibenden Schichten, welche Ports für die Adapter bereitstellen.

#### B. SOLID-Prinzipien in einer Hexagonaler Architektur

Der fundamentale Gedanke in diesem Architekturstil liegt in der eingebauten Verwendung des *Dependency-Inversion-Prinzip* zwischen dem Applikationskern und den Adaptern, da die Ports nativ dem Kern zugewiesen sind. Die Kopplung

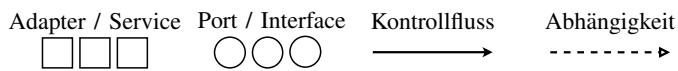
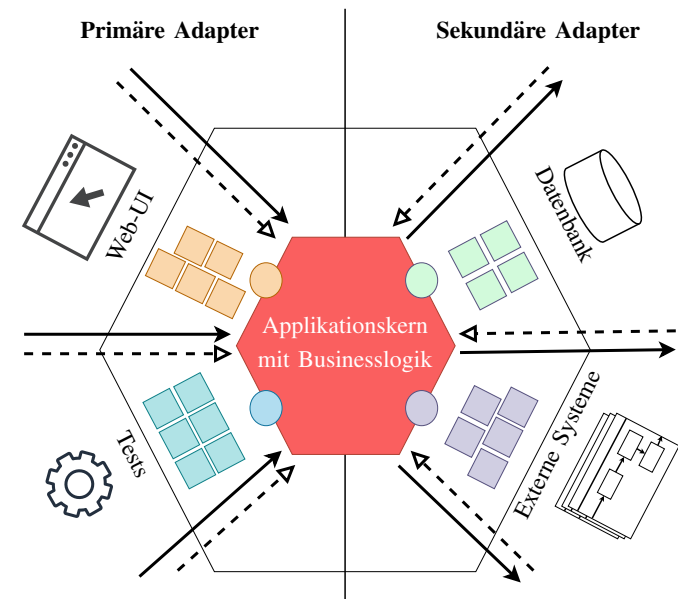


Abbildung 11. Beispielhafte Darstellung einer Hexagonalen Architektur

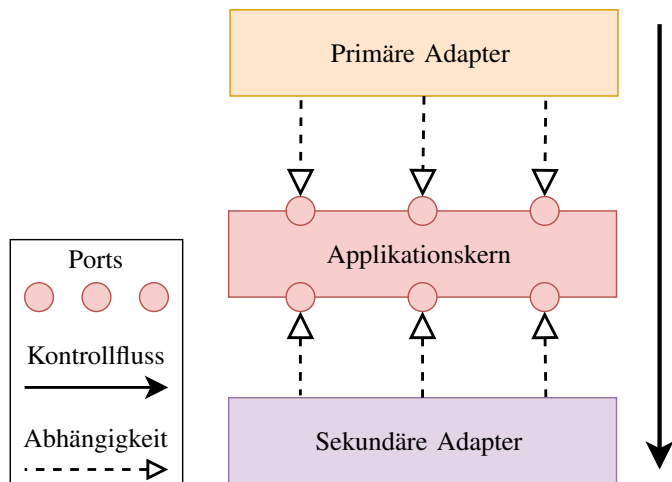


Abbildung 12. Darstellung einer Hexagonalen Architektur in Schichten

zwischen dem zentralen Teil der Software und externen Modulen wird gesenkt und die Kohäsion gestärkt. Businessanforderungen sind unter stetigen Änderungen und eine Entkopplung erlaubt Entwicklern ohne unerwarteten Nebenwirkungen diese abzuändern. Zusätzlich werden Entwickler gezwungen das *Interface-Segregation-Prinzip* zu implementieren, da die Kommunikation stets über Ports geschieht, welche als Interfaces realisiert werden. Ebenfalls wird das *Open-Closed-Prinzip* natürlich angewandt, da die Ports als geschlossen und die Adapter als offen gelten.

Hingegen ist unverändert zu der Schichtenarchitektur nur eine grobe Einteilung der Verantwortungen gegeben. Dieser Aspekt kann verbessert werden, wenn die Ports und Adapter

Architektur durch einen Domain-Driven Design Ansatz erweitert wird.

### C. Zusätzliche Entwicklungsfaktoren einer Hexagonalen Architektur

Softwareprojekte skalieren meist eher auf Basis von zusätzlichen Anforderungen und zahlreichen Sonderlocken. Da bei einem Port und Adapter Ansatz ein erhöhter Fokus auf der Businesslogik liegt, können diese Entwicklungen besser unterstützt werden und die Anwendung bleibt trotz steigender Komplexität übersichtlich und wartbar. Diese Vorteile können nur durch konstante Anwendung des *Dependency-Inversion-Prinzip* gewonnen werden. Programmieren müssen vor einer konkreten Implementierung eine passende Abstraktion designen oder diese auf benutzte Ports abstimmen. Diese Voraussetzung erhöht im Vergleich zur *Schichtenarchitektur* den Entwicklungsaufwand und die Komplexität der Architektur, kann allerdings inkorrekte Codeanpassungen verhindern.

## V. FAZIT UND VERGLEICH VON SCHICHTEN- UND HEXAGONALER ARCHITEKTUR

Aus der vorgehenden Analyse können folgende Bewertungen im Hinblick auf die SOLID-Prinzipien, Testbarkeit, Skalierbarkeit und Einfachheit gebildet werden. Hierbei wurde eine fünfwertige Skala verwendet, welche die natürliche Unterstützung des Architekturstils zur Einhaltung der jeweiligen Aspekte beschreibt. Sie reicht von hoher Unterstützung (grün) bis keine Unterstützung (rot). Diese Abbildungen sollen vor allem als Vergleich der Architekturansätze zueinander dienen.

### A. Bewertung der Schichtenarchitektur

SRP	○	○	●	○	○
OCP	○	○	○	●	○
ISP	○	○	○	●	○
DIP	○	○	○	●	○
Testbarkeit	○	○	●	○	○
Skalierbarkeit	○	○	○	●	○
Einfachheit	●	○	○	○	○

Abbildung 13. Bewertung der architektonischen Eigenschaften einer Schichtenarchitektur

### B. Bewertung der Hexagonalen Architektur

### C. Vergleich der beiden Designstrategien

Während die Schichtenarchitektur im Zentrum die verarbeiteten Daten liegen, ist der Fokus einer Hexagonalen Architektur auf den Businessanforderungen. Aus diesem fundamentalen Unterschied und der erarbeiteten Architekturanalyse ergibt




































SRP					
OCP					
ISP					
DIP					
Testbarkeit					
Skalierbarkeit					
Einfachheit					

Abbildung 14. Bewertung der architektonischen Eigenschaften einer Hexagonalen Architektur

## LITERATUR

- [1] Vaughn Vernon. *Implementing domain-driven design*. Fourth printing. Upper Saddle River, NJ: Addison-Wesley, 2015. ISBN: 9780321834577.

sich folgendes Fazit. Einerseits fällt es Entwicklern grundlegend in einer Hexagonalen Architektur die Implementierung der SOLID-Prinzipien einfacher, da diese schon tief in der Architektur durch die Einteilung in Ports und Adapter verankert sind. Vor allem gilt dies für das *Dependency-Inversion-Prinzip*. Dank einem höheren Erfüllungsgrad der SOLID-Prinzipien werden die Komponente entkoppelt und erlangen eine höhere Kohäsion. Dies wirkt sich positiv auf die isolierte Testbarkeit und Fehleranfälligkeit bei Anpassungen an der Software aus. Im Vergleich dazu entlastet eine Schichtenarchitektur nur minimal den Entwickler bei Einhaltung der SOLID-Prinzipien. Da eine Hexagonale Architektur nur eine leicht umgestellte Schichtenarchitektur darstellt, können erfahrene Entwickler dennoch auch in einem schichtenbasierten Design alle Richtlinien anwenden, indem die Kommunikation über abstrakte Schnittstellen und Invertierung der Abhängigkeiten stattfindet. Ebenfalls kann die Testbarkeit bei korrekter Implementierung genauso gewährleistet werden wie in einem hexagonalen Ansatz. Die verschiedene Denkweise einer Schichtenarchitektur verleiten Programmierer allerdings entgegen der SOLID-Prinzipien zu entwickeln, da die Datenzugriffsschicht eigentlich nur ein Detail darstellt dennoch im Mittelpunkt gerückt wird. Letztendlich ist die Wahl der Softwarearchitektur abhängig von den Anforderungen. Eine simple CRUD-Anwendung, welche kaum Businesslogik beinhalten soll, ist einfacher mit einer Schichtenarchitektur zu verwirklichen und unter Beachtung der oben genannten Designprinzipien auch wartbar, testbar und skalierbar. Ein Programm welches einen größeren Anteil an Businesslogik besitzt, wie zum Beispiel ein Kassensystem, sollte eine Hexagonale Architektur bevorzugen. Dies gewährt einen hohen Fokus auf die eigentliche Lösung der Businessanforderungen und die Skalierbarkeit der Anwendung bei korrekter Entkopplung der Komponenten.