Prinzipien und Anwendung des Softwaredesign 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. Früher wurden Anwendungen entwickelt ohne viel Wert auf Zukunftssicherheit zu legen. Sie sollten nur eine einzige Aufgabe erfüllen und wurden nur in seltenen Fällen erweitert. Allerdings haben heutzutage immer mehr Geräte eingebaute Software und die Komplexität von Sourcecode steigt mit jedem Jahr, dadurch auch die verbundenen Entwicklungskosten. Viele Unternehmen besitzen eigens geschriebene Programme, welche stets durch Marktänderungen auf neue Anforderungen anpassbar sein sollten. Über die Jahre entsteht dadurch ein verwirrendes, fragiles Konstrukt an Codezeilen, das potenziell bei jeder Änderung neue Fehler entwickeln kann, da diese Systeme nie designt waren jahrelang in Betrieb zu sein. Von diesen damals weitverbreiteten monolithischen Anwendungen sehen wir aktuell eher einen Aufschwung an kleinen, modularisierten Microservice-Architekturen. Diese bieten bei korrekter Anwendung unter anderem erhöhte Übersichtlichkeit, Skalierbarkeit, Testbarkeit und Wartbarkeit. Die Langlebigkeit der Systeme sollen durch die unterliegende Architektur, das Design der Komponenten und Softwaretests sichergestellt werden. Das unüberlegte Programmieren einer Lösung alleinig auf die jetzigen Anforderungen reicht somit in vielen Fällen heutzutage nicht mehr aus. Im Hinblick auf dieses Problem wurden verschiedene Architekturansätze und Designprinzipien entworfen, welche einen Entwickler helfen sollten eine solche stabile Anwendung zu entwickeln.

Hierzu werden in den folgenden Abschnitten einige dieser Konzepte der Softwareentwicklung erläutert. Konkret werden als Beispiele für eine Softwarearchitektur die Schichten- und Hexagonale Architektur analysiert und bewertet. Zuvor muss allerdings ein grundlegendes Verständnis über Softwaredesign geschaffen werden, indem gängige Designrichtlinien aufgezählt und durchleuchtet werden.

II. Prinzipien des Softwaredesigns

Damit eine zukunftssichere Software gewährleistet werden kann, muss die Anwendung gewünschte Eigenschaften wie unter anderem Anpassbarkeit, Wiederverwendbarkeit, Übersichtlichkeit, Skalierbarkeit, Effizienz und Korrektheit besitzen. Über die Jahre wurden verschiedene Ansätze definiert, welche bei korrekter Einhaltung diese Attribute sicherstellen sollen. Sie beziehen sich oft auf einzelne Module oder ihre Relationen zueinander. Je nach Aufgabe und Anforderungen der Softwarelösung werden verschiedenste Programmierparadigmen und unterliegende Architekturen verwendet, dabei unterscheiden sich auch die verwendeten Prinzipien. In dieser Arbeit wird der Fokus auf objektorientierte Ansätze gelegt. Dabei werden zunächst einige der Richtlinien vorgestellt, analysiert und die resultierenden Auswirkungen bei Einhaltung bzw. Nichteinhaltung erläutert. Dies soll eine Basis schaffen die Anwendung der Prinzipien zu ermöglichen und auch Architekturen sowie Sourcecode bewerten zu können.

A. Das SOLID-Akronym

Von Michael Feathers und Robert C. Martin geprägt, zählen die SOLID-Prinzipien zu dem Fundament eines stabilen Designs. Sie erlauben Software auch bei steigender Komplexität weiterhin wartbar zu bleiben. Idealerweise sollte somit eine Softwarearchitektur den Entwickler bei der Anwendung dieser Prinzipien zu unterstützen. Für die nachfolgende Analyse liegen diese fünf Konzepte im Vordergrund.

1) Single-Responsibility-Prinzip (SRP): Durch die erste Richtlinie soll sichergestellt werden, dass die Verantwortlichkeit eines Moduls zu maximal einem Akteur gehört. Konkret darf jedes Modul nur für eine Anforderung zuständig sein. Sollten mehrere neue Anwendungsfällen definiert werden, dann muss bei korrekter Einhaltung des *SRP* Komponenten maximal einmal angepasst werden. Eine allgemeinere Variante besagt, dass jede Variable, Methode, Klasse usw. genau eine Aufgabe besitzen soll. Dadurch bleiben Module übersichtlich und die Anzahl der Abhängigkeiten gering. Dies erhöht die allgemeine Testbarkeit und schützt vor unerwarteten Nebeneffekten bei Codeanpassungen. Hingegen kann bei Verletzung eine enge Bindung der Klassen entstehen und die Software dadurch monolithische Eigenschaften entwickeln.

Als Beispiel kann eine Funktion gesehen werden, welche überprüft, ob ein Passwort alle Sicherheitsanforderungen erfüllt. Die gleiche Funktion wird für sowohl Adminaccounts als auch für normale Benutzeraccounts verwendet. Anfangs sind die Anforderungen beider Passworttypen gleich und somit wird ein Passwortvalidator erstellt, welche beide Klassen benutzen (Abbildung 1).

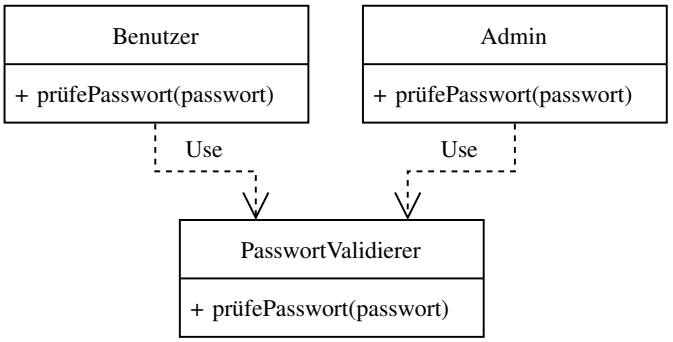

Abbildung 1. Beide Accountklassen verwenden den gleichen Validator entgegen des *SRP*

Eine neue Anforderung sieht vor, dass Adminaccounts zukünftig eine höhere Mindestlänge besitzt. Eine unaufmerksame Änderung der Funktion hat somit eine Auswirkung auf beide Benutzeraccounts entgegen des *Single-Responsibility-Prinzips*. Die Funktionen müssen somit wie in Figur 2 unterschiedliche Validatoren aufrufen.

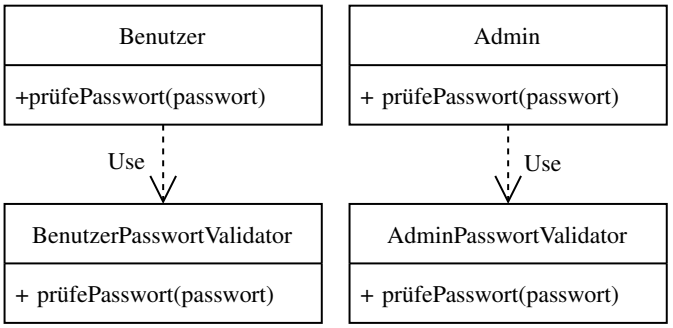

Abbildung 2. Anwendung des Single-Responsibility-Prinzip zur Passwortvalidierung

2) Open-Closed-Prinzip (OCP): Entwicklern soll es jederzeit möglich sein bereits bestehende Funktionalitäten anzupassen ohne dass der Code, welcher auf diese Funktionsweise basiert, bricht oder selbst abgeändert werden muss. Hierfür besagt das *OCP*, dass Module einerseits offen für Erweiterungen, andererseits geschlossen gegenüber Veränderungen sein sollten. Dadurch gewinnt die Applikation an Stabilität und Flexibilität. Diese zwei gewünschten Aspekte kann unter anderem erreicht werden indem ein geschlossenes, fest definiertes Interface erstellt wird. Komponente können diese Schnittstelle und die benötigten Eigenschaften implementieren und weiterhin jederzeit mit neuen Feldern und Funktionen erweitert werden. Hierbei ist die Schnittstelle geschlossen und die konkrete Implementierung offen im Sinne des *Open-Closed-Prinzip*. Durch Polymorphie ist es möglich die konkreten Implementierung hinter dem Interface auszuwechseln. Damit bleiben Abhängigkeiten austauschbar und zusätzliche Anforderungen können leichter der Software hinzugefügt werden.

Angewandt an das vorherige Passwortbeispiel ist es möglich ein Interface für den Validator zu definieren, welches die benötigte Funktion bereitstellt. Obwohl das Interface an sich als unveränderlich gilt können jederzeit weitere Validatoren hinzugefügt werden, welche die bestehenden Funktionalitäten der Abstraktion implementieren und darüber hinaus erweitern können (Abbildung 3).

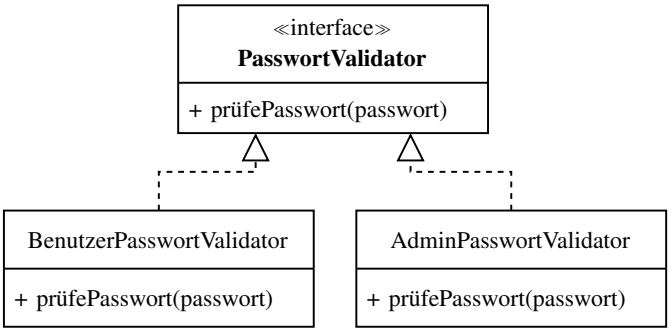

Abbildung 3. Verwenden eines geschlossenes Interface und offener Implementierungen anhand des *OCP*

3) Liskovsches Substitutionsprinzip (LSP): 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. Hierbei werden einige Bedingungen an einer Unterklasse gestellt. Unter anderem dürfen die Vorbedingungen einer Methode in einem Subtyp nicht strenger sein als in der Oberklasse und Nachbedingungen nicht schwächer erzwungen werden. Dadurch erhalten Klassen höhere Wiederverwendbarkeit und die Applikation erhält reduzierte Fehleranfälligkeit.

Als konkretes Beispiel hierfür kann die Oberklasse Benutzer (Typ T) und die Unterklasse Admin (Typ S) verwendet werden. Die Adminklasse überschreibt hierbei die vererbte Methode zum Überprüfen der Passwörter, allerdings sind die Vorbedingungen bzw. Sicherheitsbedingungen strenger. Dadurch kann die Methode ein anderes Ergebnis liefern, wenn ein Benutzertyp substituiert wird durch ein Objekt der Adminklasse, falls das übergebene Passwort zwar die Bedingungen eines Benutzerpassworts erfüllt, allerdings nicht die eines Adminpassworts (Abbildung 4).

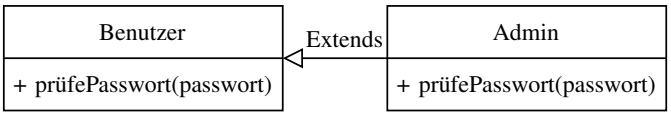

Abbildung 4. Verletzung des Liskovsches Substitutionsprinzip durch die stärkeren Vorbedingungen der Funktion in der Adminklasse

Um die Einhaltung des *LSP* zu garantieren, sollten die beiden Klassen unabhängig voneinander designt werden.

4) Interface-Segregation-Prinzip: In vor allem monolithischen Systemen finden sich öfters riesige Interfaces mit einer Vielzahl von Funktionen. Dadurch wird die Software undurchsichtig und komplex. Das *ISP* soll dies verhindern, da Clients nie gezwungen seinen sollen Schnittstellen zu verwenden, welche mehr Funktionalitäten bereitstellen als sie benötigen. Dadurch werden Interfaces übersichtlich und

abhängige Komponenten sind mehr entkoppel. Zusätzlich hilft dieses Prinzip die Einhaltung des *Single-Responsibility-Prinzip* zu gewährleisten, indem die Schnittstellen nur eine Verantwortung erfüllen. Beispielhaft bietet im Schaubild 5 ein Interface der Datenbankzugriffsschicht Funktionen zur Speicherung unterschiedlicher Entitäten an. Somit sind Service der Applikationsschicht abhängig von einem Interface mit Funktionen, welche die einzelnen Komponente nicht benötigen. Zu beachten ist ebenfalls, dass auch das *SRP* hierbei gebrochen wird.

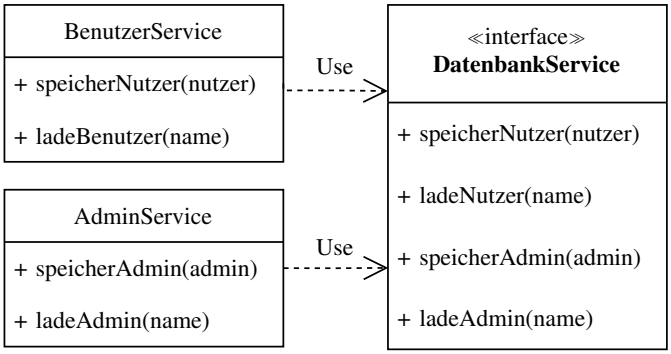

Abbildung 5. Die Abhängigkeit zwischen Klassen und einem großen Interface bricht das *ISP*

Damit eine Applikation von den oben genannten positiven Eigenschaften profitieren kann, muss die Schnittstelle durch eine Extraktion der Methoden aufgeteilt werden (Abbildung 6).

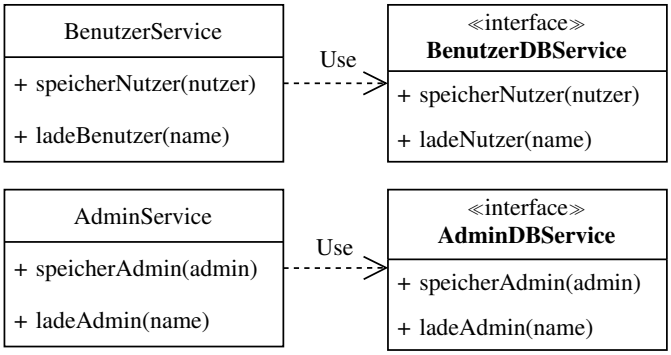

Abbildung 6. Geringere Kopplung der Service durch kleinere, spezifischere Schnittstellen

5) Dependency-Inversion-Prinzip: Um Software flexibler und anpassbarer zu gestalten, sollten die Module so viel wie möglich unabhängig von anderen Modulen designt 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 konzeptionelle höherliegende Komponenten nicht direkt von darunterliegenden Komponenten abhängig sein sollen, sondern die Kommunikation zwischen ihnen über eine abstrakte Schnittstelle geschieht. Das in der Abbildung 7 beschriebene Beispiel sollte somit die Service durch ein Interface entkoppeln, da sie auf unterschiedlichen Abstraktionsschichten liegen. Der zweite Abschnitt des Prinzips befasst sich wie diese Schnittstellen designt werden, 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. Details können hierbei die konkreten Implementierungen oder die Businesslogik sein. Bei richtiger Anwendung können dadurch höhere Module, ohne die Korrektheit des Programms zu gefährden, die darunterliegenden Module austauschen solange die Abstraktionsschicht gleich ist.

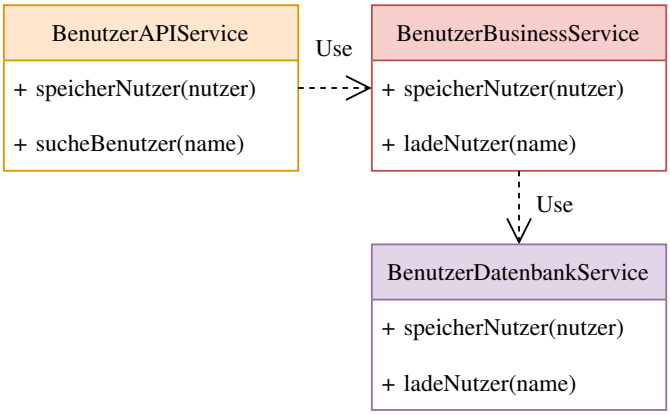

Abbildung 7. Klassen auf verschiedenen konzeptionellen Ebenen sind entgegen des *DIP* direkt voneinander abhängig

In der Abbildung 7 sind dabei beide Teile des *DIP* verletzt. Dies wird in Figur 8 korrigiert indem Details von hinzugefügten Schnittstellen abhängig gemacht werden. Das BenutzerBusinessInterface kann hierbei entweder dem BenutzerAPIService oder BenutzerBusinessService zugewiesen werden. Stehen die Businessanforderungen im Vordergrund, so sollte die Schnittstelle eher der Businessebene zugeteilt sein und die Details, somit der API-Service, davon abhängig gemacht werden.

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 architektonischem Softwaredesign sind folgende drei Konzepte von größerer Bedeutung.

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

2) Niedrige Kopplung: Abhängigkeiten zwischen Module bzw. Klassen sollte stets so gering wie möglich gehalten werden. Dadurch wird Testbarkeit, Wiederverwendbarkeit und der Schutz von äußeren Änderungen gesteigert.

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.

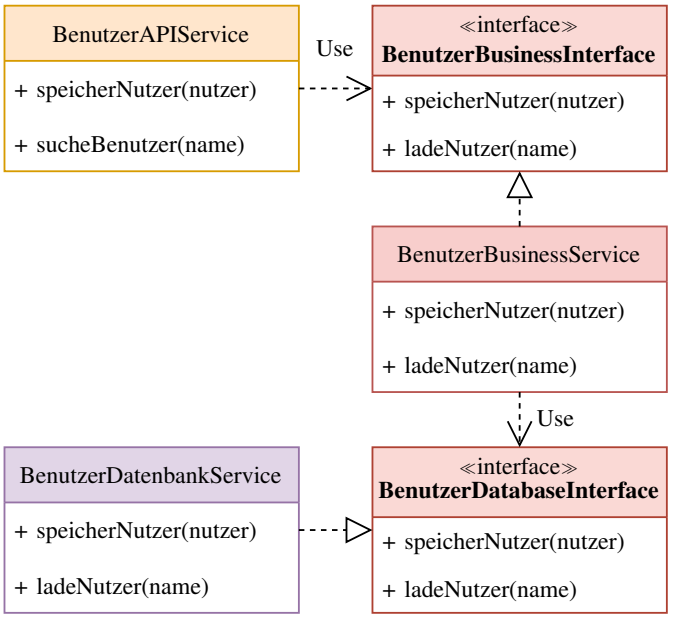

Abbildung 8. Anwendung des Dependency-Inversion-Prinzip anhand einer Schichtenarchitektur

C. Weitere Messwerte für Architekturdesign

Es gibt zahlreiche Qualitätsattribute für Software, welche je nach Kontext der Anforderungen unterschiedliche Gewichtung tragen. Die vorher aufgelisteten Prinzipen wirken sich auf diese Eigenschaften positiv aus. Für die nachfolgende Analyse sind drei Indikatoren von allgemein großer Bedeutung.

Testbarkeit definiert im welchen Maße 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 Lasten abarbeiten kann, indem einzelne Teile bzw. die ganzen Applikation weitere Ressourcen zur Verfügung gestellt bekommt. Auch wie übersichtlich und anpassbar eine Applikation auch bei steigender Anzahl an Codezeilen bleibt, spielt bei Skalierbarkeit eine Rolle. *Einfachheit* charakterisiert wie viel Erfahrung und Aufwand erfordert wird, um die gewählte Architektur korrekt umzusetzen.

III. Schichtenarchitektur

Eine Schichtenarchitektur teilt die Software 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 Softwarekontext für '**Cr**eate **U**pdate **D**elete', somit sind Anwendungen gemeint, welche Daten mit geringer bis keiner Businesslogik erzeugen, bearbeiten und löschen. Der Kern einer solchen Software sind die Daten selbst, dabei werden Module und die umliegende Architektur angepasst, um die Datenverarbeitung zu vereinfachen. Der Kontrollfluss fließt dabei von höhergelegenen in tiefere Ebenen. Ohne Anwendung des *DIP* ist der Abhängigkeitsgraph gleichgerichtet mit dem Kontrollfluss. Eine graphische Darstellung einer solchen Architektur bietet Abbildung 9.

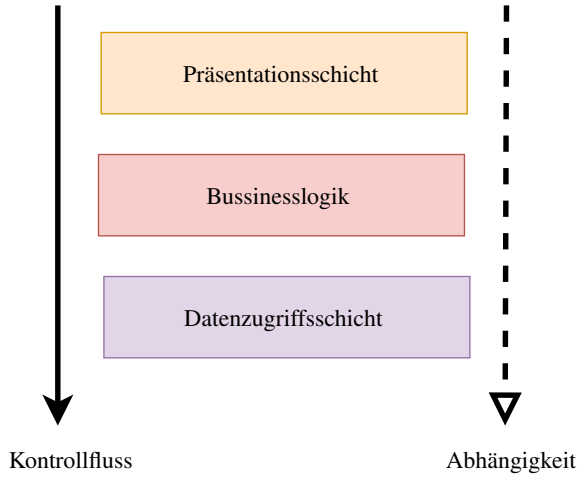

Abbildung 9. Beispielhafte Darstellung einer Drei-Schichtenarchitektur

A. SOLID-Prinzipien in der Schichtenarchitektur

Der größte Vorteil dieser Architekturart ist eine erzwungene natürliche Trennung von Funktionalitäten durch die horizontale Schichteneinteilung. Somit soll verhindert werden, dass eine Komponente beispielsweise den Zugriff auf die Datenbank und gleichzeitig Businesslogik implementiert. Allerdings ist eine vertikale Trennung nicht gegeben und Module können weiterhin verschieden konzeptionelle Aufgaben erfüllen. Dadurch ist es möglich, die Schichteneinteilung einzuhalten, jedoch das *Single-Responsibility-Prinzip* zu brechen. Dadurch ist das *SRP* nur grob in dieser Architektur verankert. Die klare Aufteilung unterstützt den Entwickler Abstraktionen zu definieren, welche Anforderungen eine höhere Schicht an darunterliegende Schichten hat. Diese können wiederum als Schnittstellen festgelegt werden, die laut dem *Dependency-Inversion-Prinzip* der verantwortlichen Schicht zugewiesen sind. Dies ist häufig die über der Implementierung liegende Ebene, kann davon aber auch je nach Applikationsfokus abweichen. In Abbildung 8 beispielsweise gelten Benutzerinterface- und Datenzugriffsschicht als Details und die Interfaces sind an die Schicht gebunden, welche die Abstraktionen hauptsächlich definieren. Damit die darüber liegenden Ebenen nicht von tiefer stattfindenden Änderungen betroffen sind, sollten diese Schnittstellen zusätzlich durch das *Open-Closed-Prinzip* als geschlossen gelten. Ebenso können die konkreten Implementierungen jedoch stets mit neuen Funktionalität erweitert oder durch andere Module ersetzt werden. Um weiterhin einen SOLID-Ansatz zu verfolgen, sind wegen dem *Interface-Segregation-Prinzip* die Schnittstellen so klein und präzise wie möglich zu designen, damit obere Schichten nur von ihren wirklich benötigten

Funktionen abhängig sind. Das *DIP*, *OCP* und *ISP* stehen eng miteinander in Verbindung. Eine Schichtenarchitektur hilft durch die Separierung der Ebenen einen Ansatzpunkt für benötigte Abstraktionen zu bilden, jedoch ist die Anwendung der Prinzipen nicht natürlich in diesem Architekturstil eingebaut.

Das *Liskovsches Substitutionsprinzip* hingegen 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.

B. Zusätzliche Entwicklungsfaktoren der Schichtenarchitektur

Um eine falsche Analyse bei fehlgeschlagenen Modultest zu vermeiden, müssen Module voneinander isolierbar sein. Dank der Schichtentrennung können die erforderliche Abhängigkeiten durch einer Verwendung von Dummy-Objekte, sogenannte Mocks, erfüllt und die geforderte Unabhängigkeit gewährleistet werden. Bei einer unzureichenden Anwendung der SOLID-Designprinzipien können einzelne Applikationsteile stark gekoppelt sein, worunter die Testbarkeit leidet 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-Prinzipen 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 Prinzipen 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.

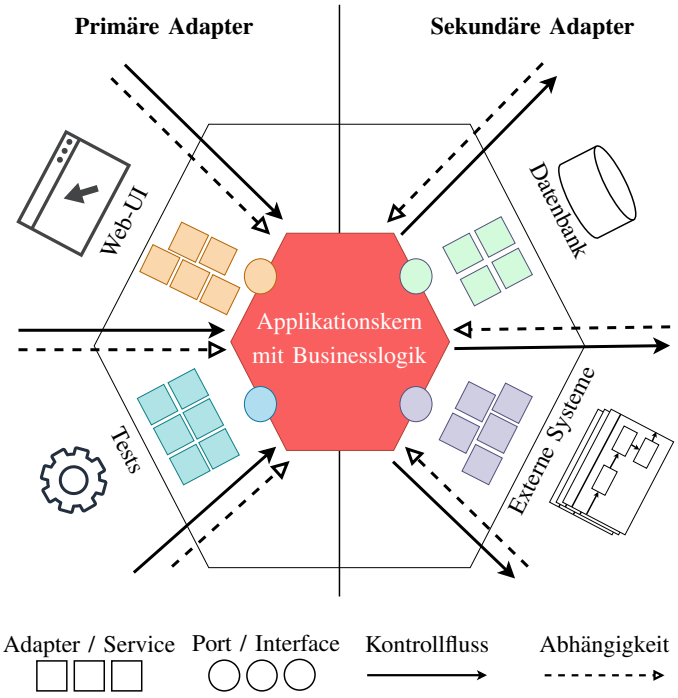

Abbildung 10. Beispielhafte Darstellung einer Hexagonalen Architektur

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.

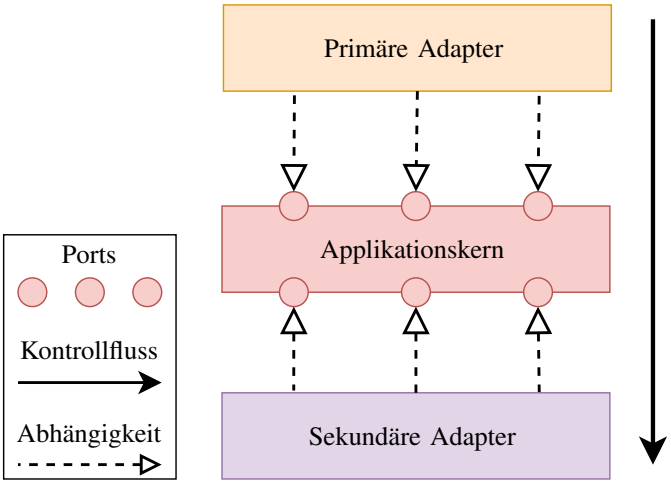

Abbildung 11. Darstellung einer Hexagonalen Architektur in Schichten

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 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 benutze Ports abstimmen. Diese Voraussetzung erhöht im Vergleich zur *Schichtenarchitektur* den Entwicklungsaufwand und die Komplexität der Architektur, kann allerdings inkorrekte Codeanpassungen verhindern.

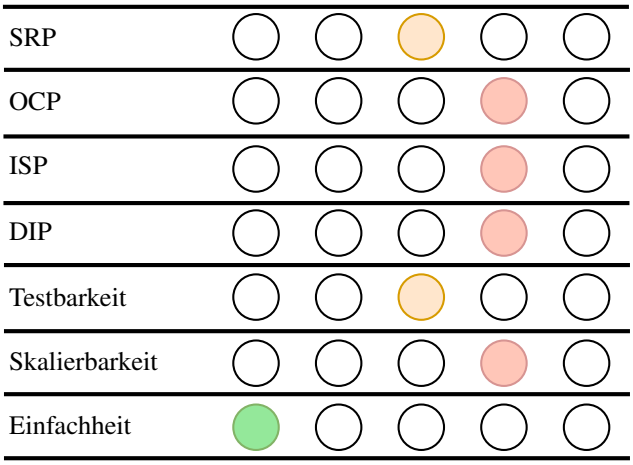

Abbildung 12. Bewertung der architektonischen Eigenschaften einer Schichtenarchitektur

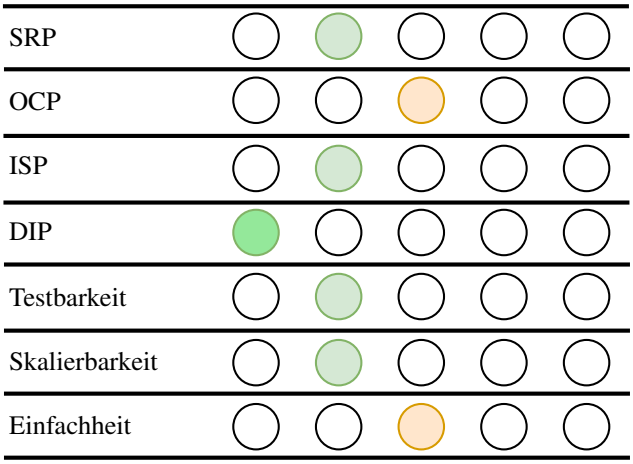

Abbildung 13. Bewertung der architektonischen Eigenschaften einer Hexagonalen Architektur

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

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 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.

LITERATUR

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