Prinzipien und Anwendung des Softwaredesigns anhand von Schichten- und Hexagonaler Architektur

Simon Thalmaier
Technische Hochschule Ingolstadt
Informatik (B.Sc.)
Matrikelnummer: 00108692

***Zusammenfassung*—Software langlebig zu designen gewann über die Jahre immer mehr an Bedeutung. Beim Implementieren von neuen Anforderungen weiterhin die Applikation übersichtlich und fehlerfrei zu halten stellt stets eine Herausforderung dar. Entwickler investieren heutzutage viel Zeit Software skalierbar, wartbar und flexibel zu gestalten. Durch die Einhaltung von etablierten Richtlinien, wie den SOLID-Prinzipien, kann ein verbessertes Softwaredesign verwirklicht werden. Somit ist das Maß an erhaltener Unterstützung bei der Implementierung dieser Prinzipien durch die gewählte unterliegende Architektur entscheidend. Die Analyse der Schichtenarchitektur zeigt, dass die SOLID-Prinzipien nur gering nativ verankert sind, jedoch eine hohe Simplizität beherbergt. Durch den grundlegenden Aufbau ist dieser Ansatz bei kleinen oder sogenannten CRUD-Applikationen zu bevorzugen. Bei hoher Priorität auf der Businesslogik ist eine Hexagonale Architektur vorteilhaft, da der Applikationskern in den Vordergrund gerückt wird. Durch die Einteilung in Ports und Adapter bei einem hexagonalen Ansatz wird die Anwendung der SOLID-Prinzipien vereinfacht. Durch die natürliche Entkopplung der Komponenten sind die Testbarkeit und Erweiterbarkeit positiv betroffen. Letztendlich ist die Auswahl der Architektur abhängig von den Anwendungsfällen und die Verwirklichung der Designprinzipien ist in beiden Ansätzen vollständig möglich.**

***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 meist 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 und 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 aus Codezeilen, das potenziell bei jeder Änderung neue Fehler entwickeln kann, da diese Systeme nie designt waren jahrelang in Betrieb zu sein. Anstatt 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. [3] [4] Die Langlebigkeit der Systeme soll 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 einem Programmierer helfen sollen 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. [11] [8] Das Akronym setzt sich aus den Anfangsbuchstaben dieser fünf Richtlinien zusammen. Sie erlauben Software, auch bei steigender Komplexität, weiterhin wartbar zu bleiben. Idealerweise sollte somit eine Softwarearchitektur den Entwickler bei der Anwendung dieser Prinzipien 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. Beim definieren neuer Anwendungsfälle sollte somit jede Komponente maximal einmal angepasst werden müssen. 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. [12] [7] [8]

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 Validator erstellt, welche beide Klassen benutzen (Abbildung 1).

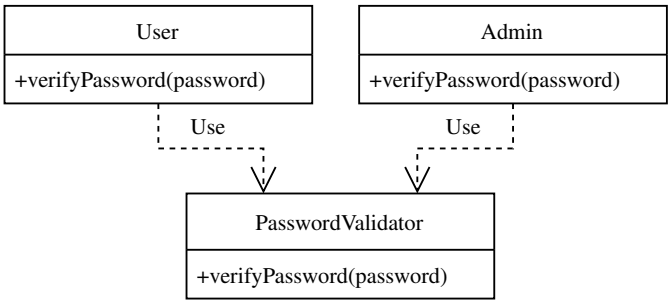

Abbildung 1. Entgegen des *SRP* verwenden beide Accountklassen den gleichen Validator

Eine unaufmerksame Änderung der Funktion hat somit eine Auswirkung auf beide Benutzeraccounts entgegen des *Single-Responsibility-Prinzips*. Eine neue Anforderung sieht vor, dass Adminaccounts zukünftig eine höhere Passwort-Mindestlänge besitzen sollen, die Grundbedingungen jedoch bleiben gleich. Die Funktionen müssen somit wie in Figur 2 unterschiedliche Klassen aufrufen. Der AdminPasswordValidator ruft hierbei die verify-Funktion des UserPasswordValidators auf, damit die Logik nicht dupliziert werden muss.

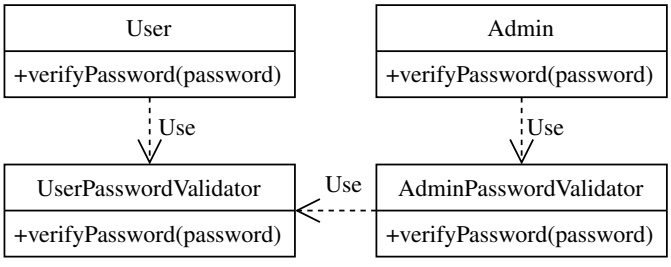

Abbildung 2. Anwendung des Single-Responsibility-Prinzip zur Überprüfung der Passwortrichtlinien

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. Komponenten 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 Implementierungen hinter dem Interface auszuwechseln. Damit bleiben Abhängigkeiten austauschbar und zusätzliche Anforderungen können leichter der Software hinzugefügt werden. [10] [8]

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).

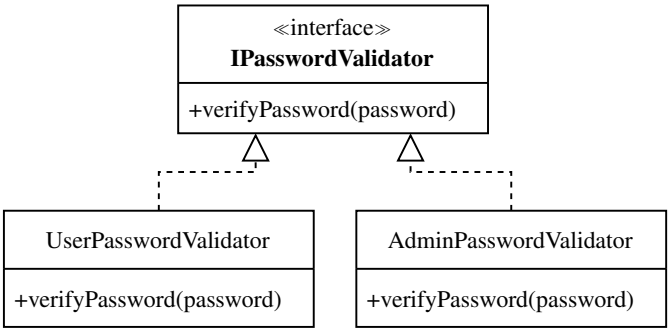

Abbildung 3. Geschlossenes Interface und offener Implementierungen anhand des *OCP*

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 die Unterklasse gestellt. Unter anderem dürfen die Vorbedingungen einer Methode in einem Subtyp nicht strenger sein als in der Oberklasse und schwächere Nachbedingungen nicht erzwungen werden. Dadurch erhalten Klassen höhere Wiederverwendbarkeit und die Applikation erhält reduzierte Fehleranfälligkeit. [6] [8]

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 durch ein Objekt der Adminklasse substituiert wird, falls das übergebene Passwort zwar die Bedingungen eines Benutzerpassworts erfüllt, allerdings nicht die eines Adminpassworts (Abbildung 4). Um die Einhaltung des *LSP* zu garantieren, sollten die beiden Klassen unabhängig voneinander designt werden.

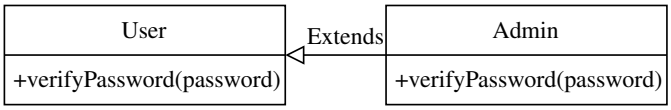

Abbildung 4. Verletzung des Liskovsches Substitutionsprinzip durch die stärkeren Vorbedingungen in der Funktion der Adminklasse

4) Interface-Segregation-Prinzip (ISP): In vor allem monolithischen Systemen finden sich oft riesige Interfaces mit einer Vielzahl von Funktionen. Dadurch wird die Software undurchsichtig und komplex. Das *ISP* soll dies verhindern, da Clients nie gezwungen sein sollen Schnittstellen zu verwenden, welche mehr Funktionalitäten bereitstellen als sie benötigen. Dadurch werden Interfaces übersichtlicher und abhängige Komponenten sind mehr entkoppelt. Zusätzlich hilft dieses Prinzip die Einhaltung des *Single-Responsibility-Prinzip* zu gewährleisten, indem die Schnittstellen nur eine Verantwortung erfüllen. [7] [8]

Beispielhaft bietet im Schaubild 5 ein Interface der Datenzugriffsschicht Funktionen zur Manipulation und Suche von Benutzern an. Ein Service, welcher Benutzer löschen soll, die sich seit langer Zeit nicht mehr eingeloggt haben, ist somit abhängig von einem Interface, welches Funktionen bereitstellt, die für diese Aufgabe gar nicht benötigt werden.

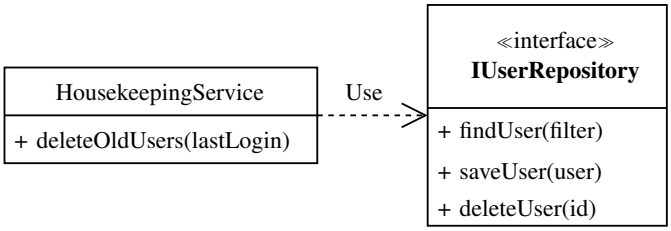

Abbildung 5. Abhängigkeit zwischen Service und einem großen Interface bricht das *ISP*

Damit eine Applikation von den oben genannten positiven Eigenschaften profitieren kann, wird die Schnittstelle durch Extraktion der Methoden in kleinere Interfaces aufgeteilt (Abbildung 6). Der HousekeepingService besitzt dadurch nur wirklich benötigte Abhängigkeiten.

5) Dependency-Inversion-Prinzip (DIP): Um Software flexibler und anpassbarer zu gestalten, sollten die Module so unabhängig wie möglich von anderen Modulen designt werden. Änderungen an dem Quelltext bergen das Risiko unerwünschte Nebeneffekte zu erzeugen oder zwingen den Entwickler auch Anpassungen an weiteren Modulen vorzunehmen. Durch eine lose Kopplung sollen solche Situationen vermieden werden. Zusätzlich ist es möglich konkrete Implementierungen hin-

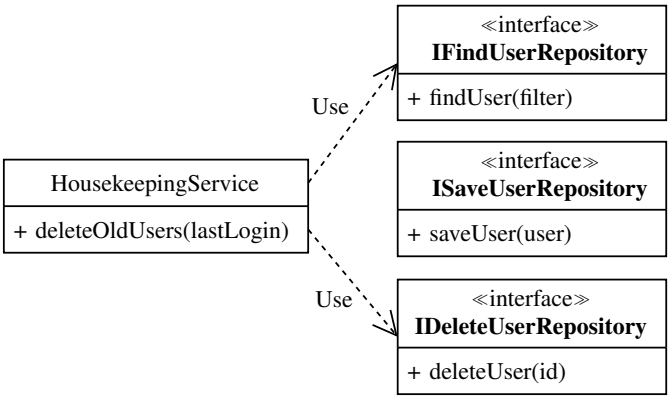

Abbildung 6. Geringere Kopplung durch kleinere, spezifischere Schnittstellen

ter den Abstraktionen ohne weitere Anpassungen auszutauschen. Dies ermöglicht das Testen von Klassen frei von ihren Abhängigkeiten, indem diese durch Dummy-Objekte substituiert werden. Der erste Teil des *Dependency-Inversion-Prinzips* besagt, dass konzeptionell höherliegende Komponenten nicht direkt von darunterliegenden Komponenten abhängig sein sollen, sondern die Kommunikation zwischen ihnen über eine Schnittstelle geschieht. Das in der Abbildung 7 beschriebene Beispiel sollte somit die Schichten durch Interfaces entkoppeln, da sie auf unterschiedlichen Abstraktionsebenen liegen. Der zweite Abschnitt des Prinzips befasst sich damit, 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 je nach Fokus der Applikation die konkreten Implementierungen oder die Businesslogik sein. Bei richtiger Anwendung können dadurch unterliegende Module ausgetauscht werden, ohne die Korrektheit des Programms zu gefährden, solange die Abstraktionsschicht gleich ist. [9] [7] [8]

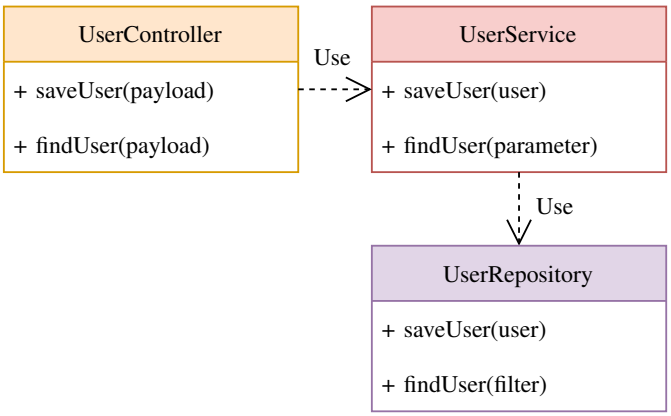

Abbildung 7. Klassen auf verschiedenen konzeptionellen Ebenen sind entgegen des *DIP* direkt voneinander abhängig

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 wird hierbei entweder dem BenutzerAPIService oder BenutzerBusinessService zugewiesen. Stehen die Businessanforderungen im Vordergrund, so sollte die Zuteilung der Schnittstelle zur Businessebene stattfinden. Die Details sind somit die API-Schicht und Datenzugriffsschicht.

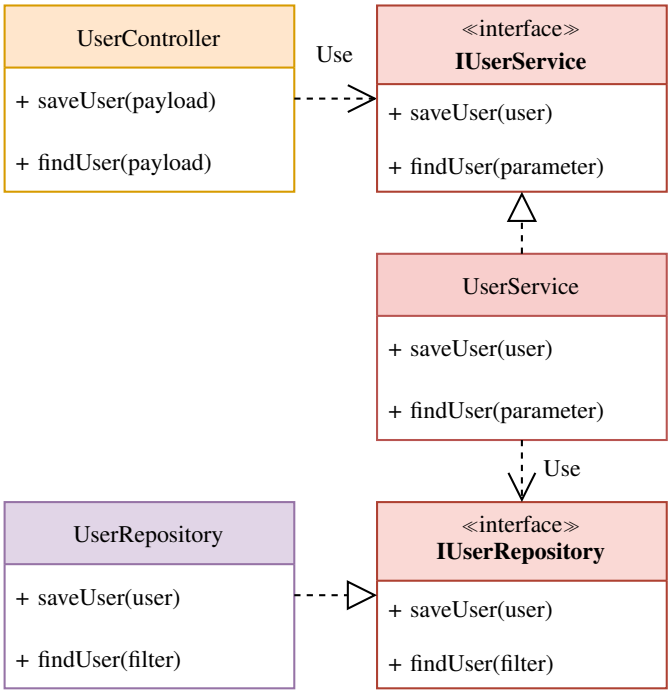

Abbildung 8. Anwendung des Dependency-Inversion-Prinzip anhand einer Schichtenarchitektur

B. Das GRASP-Akronym

Ausgeschrieben 'General Responsibility Assignment Software Patterns' ist eine Ansammlung von neun Entwurfsmustern, welche in der objektorientierten Programmierung Anwendung finden. [5] Die Prinzipien aus SOLID und GRASP beeinflussen sich teilweise gegenseitig. 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 Modulen bzw. Klassen sollte stets so gering wie möglich gehalten werden. Dadurch werden 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.

C. Weitere Bewertungskriterien

Es gibt zahlreiche Qualitätsattribute für Software, welche je nach Kontext der Anforderungen unterschiedliche Gewichtung tragen. Die vorher aufgelisteten Prinzipien wirken sich auf diese Eigenschaften positiv aus. Für die nachfolgende Analyse sind drei Indikatoren von 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. *Erweiterbarkeit* hingegen beschreibt wie übersichtlich und anpassbar eine Applikation auch bei steigender Anzahl an Codezeilen bleibt. *Erlernbarkeit* charakterisiert wie viel Erfahrung und Aufwand erfordert wird, um die gewählte Architektur korrekt umzusetzen und zu verstehen.

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 nach 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 höhere in tiefere Ebenen. Ohne Anwendung des *DIP* ist der Abhängigkeitsgraph gleichgerichtet mit dem Kontrollfluss. Eine graphische Darstellung einer solchen Architektur bietet Abbildung 9. [2] [13]

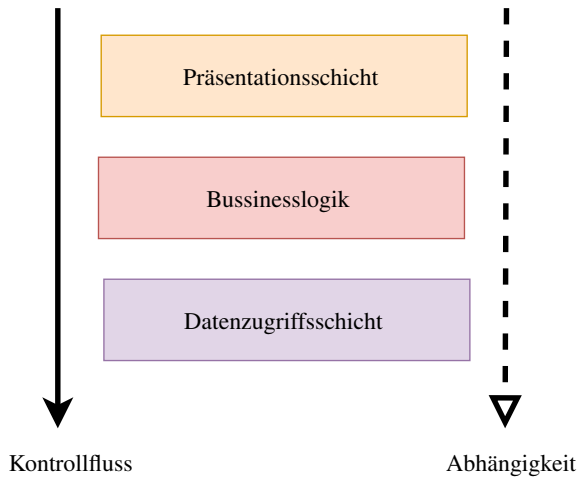

Abbildung 9. Beispielhafte Darstellung einer Drei-Schichtenarchitektur

A. SOLID-Prinzipien in einer 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 konzeptionell verschieden 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 ebenfalls Abstraktionen zwischen den Schichten zu definieren. Diese können wiederum als Schnittstellen festgelegt werden, die aufgrund des *Dependency-Inversion-Prinzips* der verantwortlichen Ebene zugewiesen sind. Dies ist häufig die über der Implementierung liegende Ebene, kann davon jedoch, abhängig von dem Applikationsfokus, abweichen. Im Diagramm 8 beispielsweise gelten Benutzerinterface- und Datenzugriffsschicht als Details und sie sind somit an die Schnittstellen, welche der mittleren Ebene zugeteilt sind, gebunden. Damit Änderungen an tiefer liegenden Schichten nicht höhere betrifft, sollten diese Schnittstellen zusätzlich durch das *Open-Closed-Prinzip* als geschlossen gelten. Ebenso können die konkreten Implementierungen weiterhin stets mit neuen Funktionalitäten 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 Schichten nur von ihren wirklich benötigten Funktionen abhängig gemacht werden. 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 Prinzipien nicht natürlich in diesem Architekturstil eingebaut.

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 Analysen vernachlässigt werden.

B. Zusätzliche Entwicklungsfaktoren einer Schichtenarchitektur

Um eine falsche Analyse bei fehlgeschlagenen Modultests zu vermeiden, müssen Module voneinander isolierbar sein. Dank der Schichtentrennung können die erforderlichen Abhängigkeiten durch Verwendung von Dummy-Objekten, 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 Schichten einteilen. Entwickler erhalten mit diesem Architekturstil eine simple, übersichtliche Methode Module und ihre Relationen zueinander zu modellieren. Die Denkweise einer CRUD-Anwendung kann jedoch 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 monolithische Struktur, bei der Codeänderungen zu unerwarteten Nebenwirkungen führen können. [13]

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 geprägten Architektur ist der Hauptgedanke die Einteilung von Komponenten in Adapter und Applikationskern. Adapter sind Schnittstellen zwischen externen Systemen und der Businesslogik, wie beispielsweise Controller, Datenzugriffsservice oder Messaging-Broker. Zusätzlich werden sie in primär und sekundär eingeteilt, wobei der Kontrollfluss von den primären Adaptern in die Businesslogik und eventuell weiter über die Sekundäradapter fließt. Somit werden primäre Adapter von außen und sekundäre Adapter durch die Applikation selbst angestoßen. Die Kommunikation zwischen Adapter und Kern geschieht hierbei über Interfaces, den sogenannten Ports, welche dem Applikationskern zugeteilt und von diesem definiert werden. Daher wird dieser Ansatz auch *Ports und Adapter Architektur* genannt. Durch diese Aufteilung soll der Fokus auf die Businesslogik gelegt werden. [1] [8]

In Abbildung 10 ist neben dem generellen Aufbau auch der Kontrollfluss und Abhängigkeitsverlauf dargestellt.

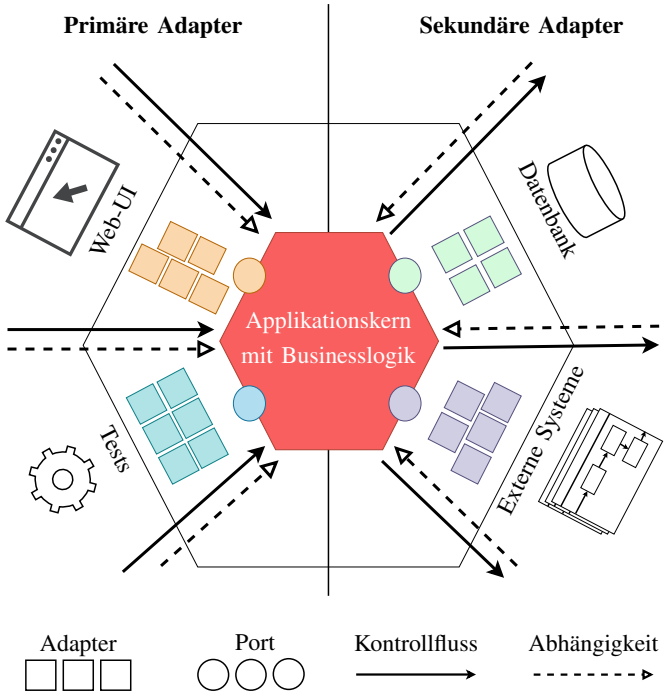

Abbildung 10. Beispielhafte Darstellung einer Hexagonalen Architektur

A. Architektonischer Vergleich mit einer Schichtenarchitektur

Bei genauer Betrachtung fällt auf, dass Hexagonale Architektur eine Variante der Schichtenarchitektur ist, welche das *Dependency-Inversion-Prinzip* fest implementiert. Hierbei werden die konzeptionell höherliegenden Schichten als primäre Adapter interpretiert und tiefer gelegene als sekundäre. Der Applikationskern besteht somit aus den verbleibenden Schichten, welche zusätzlich Ports für die Adapter bereitstellen. Jegliche Art von Businesslogik wird ebenfalls dem Kern zugewiesen. Wie in Abbildung 11 dargestellt, bildet somit die Hexagonale Architektur einen, in Hinsicht auf die SOLID-Prinzipien, verbesserten Ansatz als die herkömmliche Schichtenarchitektur.

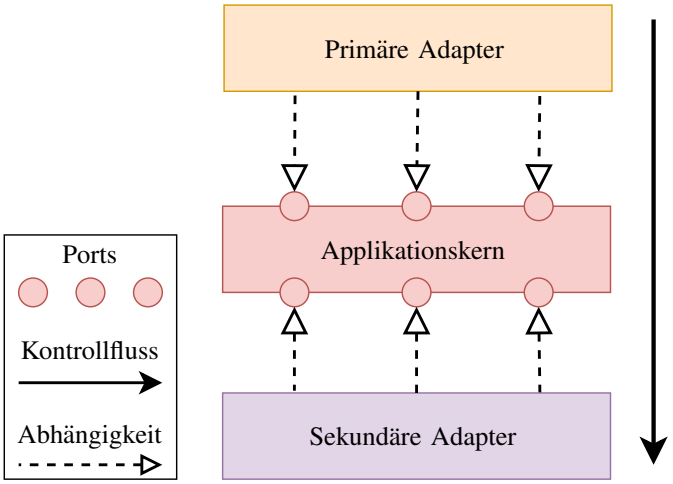

Abbildung 11. Einteilung einer Hexagonalen Architektur in Schichten

B. SOLID-Prinzipien in einer Hexagonaler Architektur

Aufgrund der Ähnlichkeiten zwischen den Architekturstilen können die Ergebnisse der vorgehenden Analyse der SOLID-Prinzipien in einem schichtenbasierten Ansatz als Grundlage verwendet werden. Ein Vorteil in diesem Architekturstil liegt in der eingebauten Verwendung des *Dependency-Inversion-Prinzips* zwischen dem Applikationskern und den Adaptern, weil die Ports nativ dem Kern zugewiesen sind und somit die Abhängigkeiten invertieren. Dank der Ports wird die Kopplung zwischen dem zentralen Teil der Software und externen Modulen gesenkt, die Kohäsion gestärkt und die Anwendung des *Interface-Segregation-Prinzips* erleichtert. Businessanforderungen unterziehen sich stetigen Änderungen und eine Entkopplung erlaubt Entwicklern ohne unerwarteten Nebenwirkungen diese anzupassen. Ebenfalls wird das *Open-Closed-Prinzip* natürlich unterstützt, da die Ports als geschlossen gelten und ihre Implementierungen erweitert werden können. Unverändert zur Schichtenarchitektur ist nur eine grobe Einteilung der Verantwortungen gegeben. Dieser Aspekt kann verbessert werden, wenn die Architektur durch *Domain-Driven Design* erweitert wird. [14]

C. Zusätzliche Entwicklungsfaktoren einer Hexagonalen Architektur

Softwareprojekte skalieren meist aufgrund von zusätzlichen Anforderungen und Spezialfällen. Da bei einem *Ports und Adapter Ansatz* ein hoher Fokus auf der Businesslogik liegt, können diese mit geringerem Aufwand implementiert werden und die Anwendung bleibt trotz steigender Komplexität übersichtlich und wartbar. Die Entkopplung der Komponenten durch Ports erhöht weiterhin ihre Testbarkeit. Codeänderungen bergen hierdurch geringeres Risiko unerwartete Nebeneffekte zu produzieren. Diese Vorteile können durch konstante Anwendung der fünf SOLID-Prinzipien maximiert werden. Die Voraussetzungen einer Hexagonalen Architektur fordern allerdings erhöhten Entwicklungsaufwand und steigern die Komplexität des Grundaufbaus.

V. Fazit und Vergleich von Schichten- und Hexagonaler Architektur

Als Ergebnis der Analyse können folgende Bewertungen im Hinblick auf die SOLID-Prinzipien, Testbarkeit, Erweiterbarkeit und Erlernbarkeit gebildet werden. Hierbei wurde eine fünfwertige Skala verwendet, welche die natürliche Unterstützung des Architekturstils zur Einhaltung der jeweiligen Aspekte beschreibt. Sie reicht von hoher (grün) bis keine Unterstützung (rot) und soll als Vergleich der Architekturansätze zueinander dienen.

A. Bewertung der Schichtenarchitektur

Die Schichtenarchitektur bietet kaum eine native Implementierung der SOLID-Prinzipien abgesehen von der Einteilung in Ebenen. Durch die geringe Entkopplung ergibt sich auch eine niedrige Erweiterbarkeit und Testbarkeit der Applikation. Durch den simplen Aufbau der Architektur gewinnt die Software an Übersichtlichkeit solange diese nicht zu komplex wird. Die Auswertung ist in Abbildung 12 dargestellt.

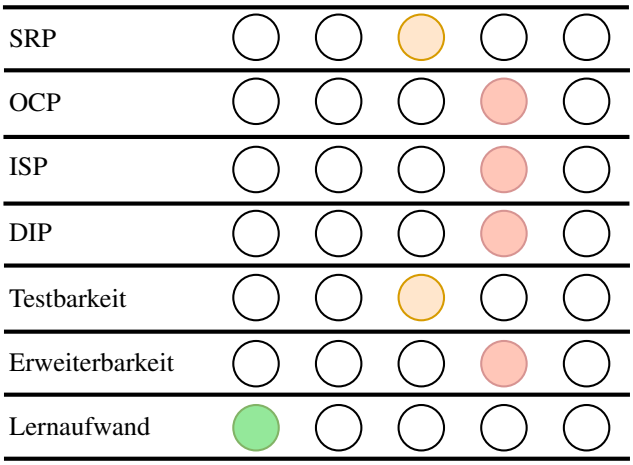

Abbildung 12. Bewertung der architektonischen Eigenschaften einer Schichtenarchitektur

B. Bewertung der Hexagonalen Architektur

Dank der Einführung von Ports erhält die Hexagonale Architektur eine natürlich Unterstützung der SOLID-Prinzipien, speziell das *Dependency-Inversion-Prinzip*. Anzumerken ist hierbei, dass auch bei diesem architektonischen Ansatz diese nicht erfüllt sein müssen. Allerdings spricht das für eine inkorrekte Anwendung der *Ports und Adapter Architektur*. Die vereinfachte Implementierung der SOLID-Prinzipien resultiert in erhöhter Testbarkeit und Erweiterbarkeit. Da konzeptionell ein hexagonaler Ansatz komplexer ist, leidet die Erlernbarkeit darunter. Das Ergebnis der Analyse wird in Abbildung 13 gezeigt.

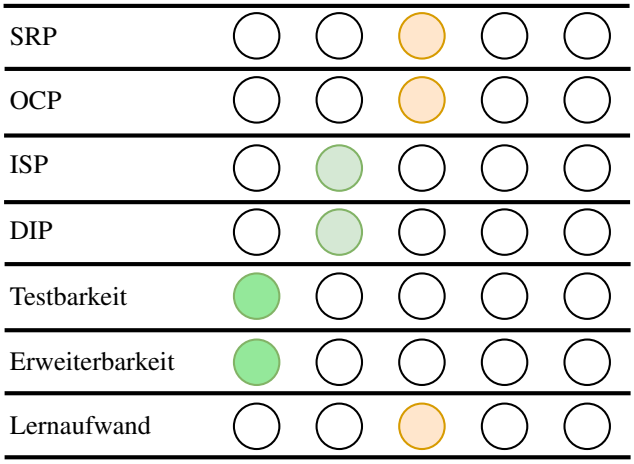

Abbildung 13. Bewertung der architektonischen Eigenschaften einer Hexagonalen Architektur

C. Vergleich der beiden Designstrategien

Während bei der Schichtenarchitektur die verarbeiteten Daten im Zentrum liegen, ist der Fokus einer Hexagonalen Architektur auf den Businessanforderungen. Aus diesem fundamentalen Unterschied und der erarbeiteten Architekturanalyse kann nachfolgendes Fazit gebildet werden. Dank einem höheren Erfüllungsgrad der SOLID-Prinzipien werden die Komponenten entkoppelt und erlangen eine höhere Kohäsion. Dies wirkt sich positiv auf die isolierte Testbarkeit und Fehlertoleranz 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 Schnittstellen und Invertierung der Abhängigkeiten stattfindet. Ebenfalls kann die Testbarkeit bei Implementierung der Designprinzipien genauso gewährleistet werden wie in einem hexagonalen Ansatz. Der simple Aufbau einer Schichtenarchitektur könnte dazu führen, dass Programmierer ohne Einhaltung der SOLID-Prinzipien entwickeln, da diese sich im Architekturstil selbst nicht als Grundbaustein widerspiegeln. Im Gegensatz hierzu müssen sich Entwickler bei einer Hexagonalen Architektur zwangsläufig mit einigen wichtigen Design-Prinzipien auseinandersetzen, um mit diesem Architekturstil arbeiten zu können. Dies fördert die nachhaltige Einhaltung der SOLID-Prinzipien und damit der Software Qualität insgesamt. Somit ist der erhöhte Lernaufwand anfangs zwar eine Hürde, aber auf langer Zeit von Vorteil. 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 ebenfalls wartbar, testbar und skalierbar. Ein Programm welches hingegen 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 Erweiterbarkeit der Anwendung bei korrekter Entkopplung der Komponenten.

Literatur

[1] Cockburn Alistair. *Hexagonal architecture*. URL: https://alistair.cockburn.us/hexagonal-architecture/.

[2] Frank Buschmann u. a. *Pattern-Oriented Software Architecture, A System of Patterns*. 1. Aufl. Wiley Software Patterns Series. s.l.: Wiley, 2013. ISBN: 978-0-471-95869-7. URL: http://gbv.eblib.com/patron/FullRecord.aspx?p=700122.

[3] Lianping Chen. „Microservices: Architecting for Continuous Delivery and DevOps“. In: *2018 IEEE International Conference on Software Architecture (ICSA)*. IEEE, 2018, S. 39–397. ISBN: 978-1-5386-6398-1. DOI: 10.1109/ICSA.2018.00013.

[4] Nicola Dragoni u. a. „Microservices: How To Make Your Application Scale“. In: *A.P. Ershov Informatics Conference (the PSI Conference Series, 11th edition)*. Moscow, Russia, 2017. URL: https://hal.inria.fr/hal-01636132.

[5] Craig Larman. *Applying UML and patterns: An introduction to object-oriented analysis and design and iterative development*. 3. ed., 12. print. Upper Saddle River, NJ: Prentice Hall, 2009. ISBN: 978-0131489066.

[6] Gary T. Leavens und Murali Sitaraman, Hrsg. *Foundations of component-based systems*. Cambridge: Cambridge Univ. Press, 2000. ISBN: 0-521-77164-1.

[7] Robert C. Martin. *Agile software development: Principles, patterns, and practices*. Alan Apt series. Upper Saddle River, NJ: Pearson Education, 2003. ISBN: 978-0135974445.

[8] Robert C. Martin. *Clean architecture: A craftsman's guide to software structure and design*. Robert C. Martin series. Boston: Prentice Hall, 2018. ISBN: 9780134494166. URL: http://proquest.tech.safaribooksonline.de/9780134494272.

[9] Robert C. Martin. *The Dependency Inversion Principle*. 1996. URL: https://web.archive.org/web/20110714224327/http://www.objectmentor.com/resources/articles/dip.pdf.

[10] Bertrand Meyer. *Object-oriented software construction*. [Nachdr.] Prentice Hall international series in computer science. New York: Prentice Hall, 1993. ISBN: 0-13-629049-3.

[11] Robert C. Martin. *Design Principles and Design Patterns*. URL: https : / / web . archive . org / web / 20150906155800 / http : / / www . objectmentor . com / resources/articles/Principles_and_Patterns.pdf.

[12] Robert C. Martin. *SRP: The Single Responsibility Principle*. URL: https : / / web . archive . org / web / 20150202200348 / http : / / www . objectmentor . com / resources/articles/srp.pdf.

[13] Sylvia Fronczak. *Layered Architecture: Still a Solid Approach*. URL: https : / / blog . ndepend . com / layered - architecture-solid-approach/.

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