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

Simon Thalmaier
Technische Hochschule Ingolstadt
Informatik
Matrikelnummer: 00108692

***Zusammenfassung*——TODO**
***Index Terms*—architecture, ports and adapters, layer, software**

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 'guten' Software Designs unterscheiden sich je nach Programmierparadigmen und unterliegender Architektur. Hierbei spielen verschieden Faktoren eine Rolle und es wurden über die Jahre einige Richtlinien definiert, welche einen Entwickler behilflich sein sollen eine solche 'gute' Software zu erstellen.

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. 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. Diese Funktion wird für sowohl Adminaccounts als auch für normale Benutzeraccounts verwendet. 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. Eine allgemeinere Variante besagt, dass jede Variable, Methode, Klasse usw. genau eine Aufgabe besitzt.

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 auszubauen ohne dass der Code, welcher auf diese Funktion basiert, abändern zu müssen. Eine Möglichkeit dieses Prinzip anzuwenden ist die Verwendung eines Interfaces, um Module mit unterschiedlichen Implementierungen

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 strengeren 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.

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 sollen Schnittstellen zu verwenden, welche mehr Funktionalitäten bereitstellen als benötigt. Dadurch sollen Interfaces übersichtlicher und vor allem nach dem *Single-Responsibility-Prinzip* auch nur eine Verantwortung erfüllen.

5) Dependency-Inversion-Prinzip: Um die Software flexibler und anpassbarer zu gestalten, sollten die Module so weit 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* befasst sich hiermit. Der erste Teil dieser Richtlinie besagt, dass höherliegende Komponenten nicht direkt 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.

B. Das GRASP-Akronym

Ausgeschrieben 'General Responsibility Assignment Software Patterns' ist eine Ansammlung von neun Entwurfsmustern, welche in der objektorientierten Programmierung Anwendung finden.

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.

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. Weiter Indikatoren

Es gibt viele weitere Qualitätsattribute von Software, welche je nach Anwendung unterschiedliche Gewichtung tragen. Universal sind *Testbarkeit*, *Skalierbarkeit* und *Einfachheit* Eigenschaften von hoher Bedeutung bei Softwarearchitekturen.

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.

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 verschieden Aufgaben erfüllen. Dadurch ist es möglich, die Schichteneinteilung nicht zu verletzten, 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.

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

B. Weitere Entwicklungsfaktoren

Durch die Schichtentrennung können einzelne Module unabhängig voneinander getestet werden indem Abhängigkeiten durch Dummy-Objekte, sogenannte Mocks, ersetzt werden.

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. Adapter sind Schnittstellen zwischen externe Systeme und der Businesslogik. Daher wird dieser Stil auch *Ports und Adapter Architektur* genannt.

A. Vergleich mit Schichtenarchitektur

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

B. Designprinzipien 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. 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.

Unverändert zu der Schichtenarchitektur ist 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. Weiterhin ist das *Liskovsches Substitutionsprinzip* nicht in der Architektur verankert.

C. Vergleich von Schichten- und Hexagonaler Architektur

LITERATUR

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