Ports and Adapters – Architektur für moderne Applikationen

Klassische Software-Architektur ähnelt oft einem Schichtensalat. Doch das, was Sie auf eine Party mitbringen würden, eignet sich nicht als Basis für moderne Anwendungen.

BENJAMIN KLÜGLEIN *

Er ist auf Partys meist ein gern gesehener Gast. Optisch macht Eindruck, wie sich der Mais an die Mayonnaise schmiegt, wie der Sellerie mit dem Kochschinken eine Symbiose eingeht, und doch jede Ebene verlockend getrennt ist. Dank seines meist gläsernen Gefäßes kann ihn jeder von Außen bewundern - den Schichtsalat!

Am nächsten Morgen jedoch offenbart sich ein anderes Bild. Aus den einst fein säuberlich eingezogenen Schichten ist eine klumpige Masse geworden. Was einmal appetitlich und einladend aussah, erinnert nur noch vage an den Star des gestrigen Buffets.

Wer Anwendungen entwickelt kennt vielleicht ein ähnliches Problem. Was vor Wochen und Monaten noch eine gute Architektur schien und deutlich als einzelne Schichten abzeichnete, erscheint plötzlich in einem ganz anderen Licht. Es fällt schwer die einzelnen Schichten noch als solche zu erkennen. Zu sehr sind Details aus der Geschäftslogik in die Darstellung eingeflossen. Die Wahl der Datenbank bestimmt über die Funktionalitäten. Die Anwendung lässt sich nur unter Schwierigkeiten in Teilen, geschweige denn in ihrer Ganzheit testen. Man stellt sich die Frage, was ist noch Mais, was ist schon Datenbank?

Mit „Ports and Adapters“ hat Alistair Cockburn ein Architekturmuster definiert, das genau diesen Problemen begegnen soll. Der vorliegende Artikel beschreibt seinen Einsatz in der Praxis.

* Benjamin Klüglein
... ist Senior Software Engineer bei Method Park

Houston, haben wir ein Problem?

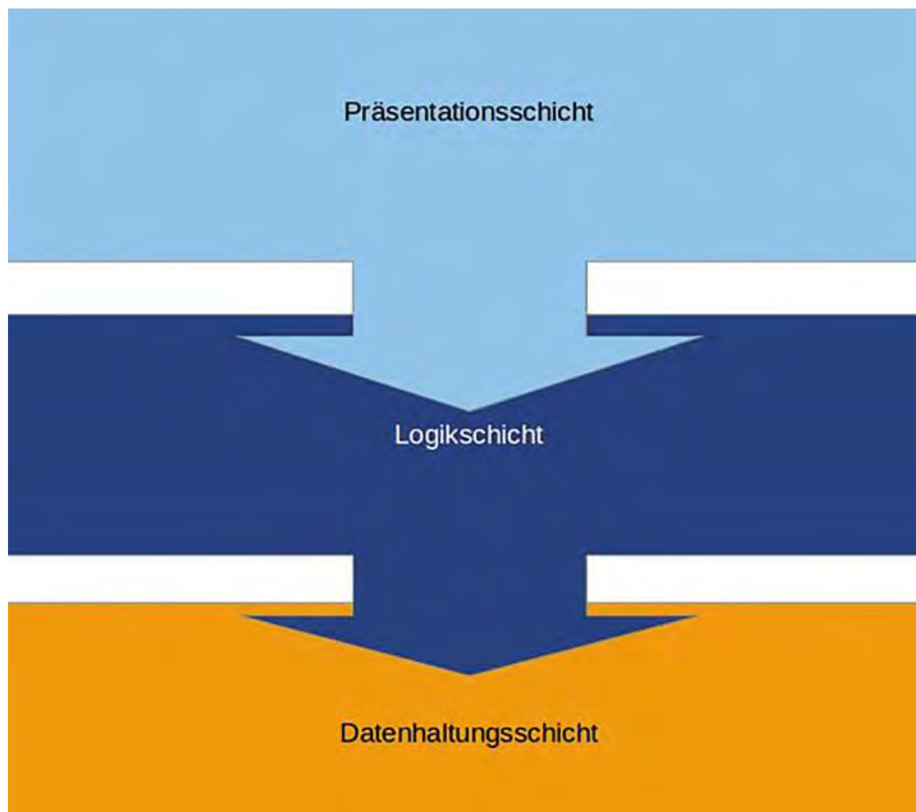
Sind in unserem Salat-Beispiel die Probleme mehrheitlich ästhetischer Natur, so zeigen sich bei Anwendungen nach einem Schichtenmuster handfeste Probleme. Zunächst wollen wir die klassische Schichtenarchitektur betrachten. Sie besteht zumeist aus drei Schichten:

- Der Datenhaltungsschicht, die für die Speicherung der Anwendungsdaten zuständig ist,
- der Logikschicht, die das Herzstück der Anwendung darstellt,
- und zuletzt der Darstellungsschicht, zuständig dafür die Daten anzuzeigen und mit dem Benutzer zu interagieren.

Um die drei Teile der Anwendung nun koordiniert miteinander kommunizieren zu lassen, legt man meist fest, dass eine Schicht nur mit der unter ihr liegenden Schichten kommunizieren darf. In der Praxis werden jedoch häufig Geschäftslogik und Benutzeroberflächen-Code vermischt. Daraus resultieren folgende Hauptproblemfelder:

- Die Anwendung kann nicht ohne größeren Aufwand automatisiert getestet werden.
- Es ist knifflig die Anwendung ganz oder in Teilen wiederzuverwenden beziehungsweise zu ersetzen.
- Die Entwicklung der einzelnen Anwendungskomponenten lässt sich nur schwer unabhängig voneinander vorantreiben.

Hinzu kommt, dass Anwendungen häufig stark an eine Datenbank, eine Darstellungsform (wie HTML) oder einen externen Service gekoppelt sind. Ändert sich das Datenbank-



Bilder: Method Park

Klassische Schichtenarchitektur: *Idealvorstellung; jede Schicht kennt nur die jeweils darunterliegende.*

schema oder muss die Datenbank gar durch eine andere ersetzt werden, können Entwickler in ihrer Arbeit blockiert werden und so unnötige Kosten entstehen.

Ein Beispiel aus der Praxis: Eine Anwendung des Kunden nutzte einen proprietären SQL-Datenbank-Server für die Datenhaltung. Für bestimmte Anwendungsfälle wurde die Funktionalität als sogenannte „Stored Procedures", also als in der Datenbank hinterlegte Funktionen, realisiert. Wann immer ein Benutzer gewisse Anwendungsfälle durchführen wollte, rief er von der Darstellungsschicht über die Persistenzschicht eine entsprechende Prozedur auf. Die Anwendung war somit direkt abhängig von der gewählten Persistenzlösung. Details aus der eigentlich untersten Schicht übertrugen sich bis fast hinauf in die oberste.

Zunächst stellte sich dieser Umstand nicht als Problem dar. Doch nach Jahren der Entwicklung änderten sich die Anforderungen an die Software. Die Anwendung sollte nun vom Datenbank-Server unabhängig sein. Bisher musste zu jeder Installation der Software auch der Datenbank-Server auf dem Rechner installiert werden. Das führte dazu, dass die Rechner mit der entsprechenden Leistung - ausreichend für Datenbank und Anwendung - ausgelegt sein mussten. Günstige Laptops waren demnach keine Alternative. Zudem ging es nicht zuletzt darum Lizenzkosten für jede Installation einzusparen.

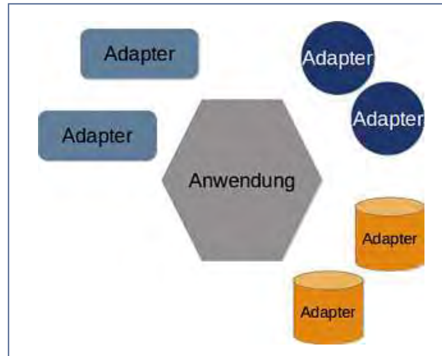
Durch die Vermischung von Anwendungslogik und Datenhaltung ließ sich die Persistenzschicht nicht ohne Weiteres auszutauschen. Zur Lösung des Problems wurde ein Service entwickelt, der die Interaktion mit der Datenbank übernahm.

Wäre die Anwendung ursprünglich nach dem „Ports and Adapters" Muster entwickelt worden, hätte man nicht in eine komplett neue zusätzliche Anwendung investieren müssen. Ein einzelner neuer Adapter wäre ausreichend gewesen.

Im folgenden wird das Prinzip der „Ports and Adapters" genannten Architektur in vier Schritten näher erläutert.

Definieren von Adaptern und Ports

Anstatt der üblichen Schichten wird die Anwendung in die Namen gebenden Ports und Adapter eingeteilt. Adapter sind Komponenten nach dem klassischen Adapter-Pattern der Gang of Four. Das Wort „Port" wurde gewählt, um an die Ports eines Computers zu erinnern. An einen solchen Port kann ein beliebiges Gerät angeschlossen werden. Dazu muss es lediglich das Protokoll des Anschlusses verstehen. Für jedes Gerät



Hexagon: *Die Anwendung ist das zentrale Bauteil und bietet Ports für eine Vielzahl von Adaptern.*

gibt es einen Adapter, der zwischen der API und den Signalen übersetzt, die das Gerät benötigt. Ein passendes Beispiel hierfür sind die USB-Anschlüsse an einem Rechner. Von Abschussrampen die Schaumstoffpfeile verschießen, bis zu Tastaturen und Mäusen kann man dank einheitlicher Schnittstelle alles anschließen und betreiben.

Dieses Bildnis aus Anschlüssen und Adaptern lässt sich leicht auf Teile von Anwendungen übertragen: Die Benutzeroberfläche (GUI - Graphical User Interface) ist ein Beispiel für einen Adapter, der die Kommunikation zwischen Nutzer und Anwendung ermöglicht. Eine Datenbank ist ein Adapter, der die Datenhaltung verwaltet.

Eine Anwendung nach dem „Ports and Adapters"-Muster lässt sich generell wie folgt darstellen: Ein Sechseck wird verwendet, um zu verdeutlichen, dass eine Innen- und Außenasymmetrie besteht und dass verschiedene, in ihrer Funktion ähnliche Ports gibt. Es soll zudem darstellen, dass es eine Anzahl unterschiedlicher Ports vorhanden ist. Dabei geht es nicht um die Zahl Sechs im Speziellen. Vielmehr erhalten Entwickler beim Entwurf Platz, um die verschiedenen Ports und Adapter einzuzeichnen. Der alternative Name des Musters ("Hexagonale Architektur") leitet sich von dieser Darstellung ab.

Bei Ports werden primäre und sekundäre Ports unterschieden. Primäre Ports sind solche, die die Anwendung anbietet und von außen aufgerufen werden. Die eigentliche Anwendungslogik ist z.B. ein primärer Port.

Sekundäre Ports werden von der Anwendung selbst aufgerufen. Der Port für die Datenhaltung ist ein solcher sekundärer Port.

Vorgehen und Vorteile von „Ports and Adapters"

Beim Entwurf von Anwendungen nach dem „Ports and Adapters"-Muster beginnt man üblicherweise mit der Definition der einzelnen Ports. Eine „klassisch aufgebaute"

Anwendung mit Datenhaltung, Geschäftslogik und Benutzeroberfläche besitzt zunächst, vereinfacht dargestellt, zwei Ports: zum einen den Benutzer- und zum anderen den Datenhaltungsport.

Der Benutzer soll die Anwendung durch eine HTML-Oberfläche steuern können. Anderen Clients bzw. Tests sollen die Möglichkeit haben unsere Anwendung mittels einer REST-Schnittstelle zu steuern. Die Daten der Anwendung sollen in einer Postgres-Datenbank gespeichert werden können. Und zu Testzwecken soll es möglich sein die Anwendung ohne eine Datenbank zu betreiben.

Daraus ergeben sich effektiv vier Adapter, die Sie der entsprechenden Abbildung entnehmen können (siehe „Systermaufbau im Überblick“). Im Anschluss an die Definition der Adapter und Ports werden deren Schnittstellen beschrieben. Dies kann entweder in UML geschehen oder direkt durch die Erstellung von Interfaces, wie man sie aus C# oder Java kennt.

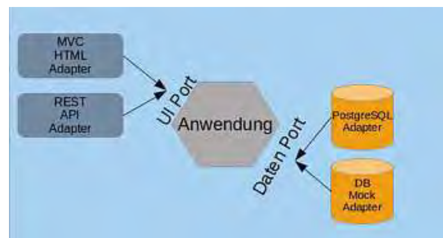
Nach dem Entwurf der einzelnen Ports beginnt man mit der Implementierung. Entscheidender Vorteil: Die Implementierung der Adapter kann ab diesem Zeitpunkt parallel erfolgen, denn die Subsysteme der Anwendung kommunizieren nur über die zuvor definierten Ports. Daraus ergeben sich folgende Vorteile:

1. Klare Einteilung der Komponenten: Der Einsatz des Patterns führt zu einer eindeutigen Einteilung der unterschiedlichen Komponenten. Alle Objekte, die beispielsweise für die Darstellung der Anwendung gebraucht werden, existieren lediglich im entsprechenden GUI-Adapter, während die für die Kommunikation mit der Datenbank notwendigen Objekte nur dem Datenbankadapter bekannt sind.

2. Eine Anwendung in mehreren Ausprägungen: Ähnlich wie in einem Baukastenprinzip lassen sich in der Hexagonalen Architektur verschiedene Varianten einer Anwendung zusammenfügen. So wäre es denkbar den Postgres-Datenbank-Adapter durch einen Android-SQLite-Adapter und den HTML-UI-Adapter durch einen nativen Android-UI-Adapter zu ersetzen. Mit dem Austausch von lediglich zwei Komponenten wandelt man die Anwendung zu einer nativen Android-Anwendung, ohne dass die eigentliche Geschäftslogik geändert werden muss.

3. Parallele Entwicklung: Die klare Einteilung in Komponenten und in wohldefinierten Schnittstellen ermöglicht es verschiedenen Teams oder Einzelpersonen parallel an der jeweiligen Implementierung zu arbeiten. Zum Zeitpunkt des Entwicklungsstarts ist es einfach das Verhalten noch fehlender Komponenten mit „Test-Adaptern“ zu simulieren. So muss man nicht auf die Fertigstellung etwa der Datenhaltungsschicht warten.

Vereinfachte Darstellung: *Eine Anwendung mit zwei Ports.*



Systemaufbau im Überblick: *Eine klassische Anwendung mit UI und Datenbank im Stil der hexagonalen Architektur*

4. Austauschbarkeit der Adapter: Sollte es notwendig sein Funktionalität auszutauschen, beispielsweise das Datenbanksystem zu ersetzen, darf sich aus Anwendungssicht nicht zwangsweise auch das Protokoll zwischen der Anwendung und der Datenhaltungskomponente ändern. Durch den Einsatz von „Ports and Adapters“ kann man dieses Risiko minimieren. Adapter sind leicht austauschbar; daher ist ein Wechsel zwischen Datenbanken nichts weiter als ein Wechsel des entsprechenden Adapters.

5. Einfache Erweiterbarkeit und Skalierbarkeit: Durch das konsequente Entwickeln gegen Ports lässt sich die grundlegende Adapterstruktur leicht ändern. So wäre es denkbar die einzelnen Adapter des Systems als Microservices zu realisieren. Ohne größeren Aufwand ist es so möglich eine Anwendung von einem lokalem zu einem verteilten System zu migrieren.

6. Keine Festlegung auf eine Zulieferform: Da die Hexagonale Architektur keinerlei Annahmen über die Darstellung der Daten und die Entgegennahme der Benutzereingaben trifft, ist man folglich an dieser Stelle auch nicht festgelegt. Hat man seine Anwendung ursprünglich als schichtenbasierte Webanwendung entworfen, so ist es mitunter schwer ihre Funktionalität einer mobilen Anwendung zur Verfügung zu stellen. Im Falle der Hexagonalen Architektur stellt die mobile Anwendung nur einen weiteren Adapter für die Entgegennahme der Benutzereingaben dar.

7. Keine Festlegung auf eine Architekturform innerhalb der einzelnen Komponenten: Innerhalb der Hexagonalen Architektur werden keinerlei Annahmen über den Entwurf der einzelnen Adapter getroffen. Um beispielsweise eine webbasierte Oberfläche als Adapter für die Benutzerinteraktion zu entwickeln, kann man sich problemlos eines Musters für die Oberflächenentwicklung, wie etwa MVC (Model-View-Controller), bedienen.

Höhere Komplexität und Orchestrierung der Anwendung

Wie jede Architekturentscheidung bringt auch die Hexagonale Architektur Nachteile mit sich. Zum Einen resultiert die Implementierung in einer höheren Komplexität: Durch den Einsatz jeder Abstraktion erhöht sich zwangsläufig die Komplexität der betroffenen Anwendung. Anders als bei einer monolithischen Anwendung befinden sich die einzelnen Komponenten des Gesamtsystems mitunter nicht mehr an einem Ort. Möglicherweise sind sie über mehrere Projekte und Subprojekte verteilt, was dazu führt, dass sich neue Kollegen zunächst mit der Orientierung etwas schwerer tun.

Zum Anderen wird eine Orchestrierung der Anwendung notwendig. Um die einzelnen Teile der Anwendung zusammenzufügen, bedarf es einer Komponente für diese Aufgabe. Diese kann ihrer Aufgabe manuell oder per Dependency Injection nachkommen. Letzteres wiederum erhöht die Komplexität zusätzlich.

Die Hexagonale Architektur hat wie jedes Architekturmuster ihren Preis und mag sich für sehr kleine Projekte nicht lohnen. Ist jedoch abzusehen, dass es sich bei einer neu zu entwickelnden Anwendung um mehr als nur einen Prototyp handelt oder dass sie über einen langen Zeitraum im Einsatz sein wird, lohnt sich die Investition sicherlich.

Erfahrungsgemäß eignet sich das Muster in der Praxis hervorragend und bereitet dem Schichtensalat erfolgreich ein Ende. Nicht nur lassen sich Anwendungen ganzheitlich und in Teilen sehr einfach testen, das Muster ist auch eine echte Investition in die Wart- und Erweiterbarkeit der zu entwickelnden Anwendung.

Man möchte eines schönen Tages seine bisherige Desktopanwendung auch im Web, auf Smartphones oder Tabletts anbieten? Wohl dem, dessen Anwendung nach „Ports and Adapters“ entwickelt wurde - die brandneue mobile Anwendung ist nur wenige neue Adapter entfernt. // SG

Method Park

Der Beitrag ist urheberrechtlich geschützt. Bei Fragen zu Nutzungsrechten wenden Sie sich bitte an pdf@vogel.de