# Architekurmuster

## Schichten (Layer) (R1)

Schichten (Layers, Tiers) zerlegen Systeme in Benutzungshierarchien. Jede Schicht bietet dabei den darüberliegenden Schichten Dienste an und verwendet ihrerseits Dienste der darunterliegenden Schichten.

Viele moderne Entwicklungsframeworks basieren auf Schichtenmodellen (u. a. Java-Enterprise, Microsoft.NET). Intensive Anwendung finden Schichten auch im domänenzentrierten Design nach [Evans

03], dessen Vorschlag für die wesentlichen vier Schichten Sie in der

folgenden Abbildung sehen.

Interaktive Informationssysteme basieren sehr häufig auf diesem Modell. Es ermöglicht eine saubere Trennung zwischen Fachlichkeit und Technik, bei der eine einzige Schicht (der *Domain Layer* ) sämtliche

fachlichen Bestandteile zusammenfasst.



Quelle: Architekturmuster kompakt von Gernot Starke

Eigene Bemerkung: Ich kenne die Schichtenarchitektur in einer strengeren Variante. Eine Schicht ruft immer nur die direkt unterliegende Schicht auf. In dieser Version muss die Schicht User Interface, alle Unterschichten kennen. Dies erschwert die Austauschbarkeit einzelner Schichten, hat aber einen Vorteil in der Performance.

## Pipes und Filter (R1)

Bausteine nehmen an Eingangsschnittstellen fortlaufend Daten entgegen, wenden darauf – gemäß ihrem Verwendungszweck – eine Funktionalität an und übergeben die derart transformierten Daten ihrer Ausgangsschnittstelle. Solche Bausteine heißen *Filter* , und die

Übertragungskanäle zwischen Filtern nennt man *Pipes* . Filter arbeiten komplett eigenständig, ohne mit anderen Filtern irgendwelchen Verarbeitungszustand auszutauschen. Sie kennen die vor und nach

ihnen arbeitenden Filter nicht.

Das Pipes and Filter Muster besteht i.a. aus 4 verschiedenen Arten von Klassen:

* Datenquelle (englisch Data Source)
* Datensenke (englisch Data Sink)
* Filter
* Pipe

**Verantwortung eines Filters**

Ein Filter erhält Daten von einer Pipe, bearbeitet diese Daten und gibt die bearbeiteten Daten wieder.

**Verantwortung einer Pipe**

Eine Pipe erhält Daten von der Datenquelle oder vom vorherigen Filter, und transportiert diese Daten zum nächsten Filter, oder zur Datensenke. Die Aufgabe sind Daten zu transportieren, diese ggf. zu puffern und 2 Nachbarn zu synchronisieren.



Abbildung Zusammenspiel zwischen Pipes und Filter

## Client/Server (R1)

Client Server Systeme bestehen aus mindestens zwei unabhängigen, miteinander kommunizierenden Prozessen:

Clients benötigen bestimmte Dienste oder Services, die sie bei einem bekannten Server erfragen.

Server stellen Clients bestimmte Dienste bereit. Auf entsprechenden Anfragen hin erfüllen sie diese Aufgaben und schicken über das jeweilige Netzwerk die Ergebnisse an den aufrufenden Client zurück.

Die Kommunikation zwischen Client und Server erfolgt synchron – ansonsten klassifiziert man diese Systeme als ereignis- oder nachrichtenbasiert.

Quelle: Effektive Softwarearchitekturen 8. Auflage Kapitel 4.5.4.1

## Adapter und Facade (R1)

Wenn Sie ein bereits existierendes Modul verwenden wollen, dessen Schnittstelle allerdings inkompatibel zu ihrer benötigten Schnittstelle ist, kann das Adapter-Muster helfen.

Der Adapter, auch bekannt als Wrapper, stammt vom englischen »to adapt«,

was so viel heißt wie an- oder einpassen. Der Adapter bewirkt eine Anpassung der Schnittstellen.



Abbildung Adapter werden benutzt, falls Schnittstellen nicht zueinander



Abbildung Kapselung des Zugriffs auf Fremdsysteme durch ein Fassadenobjekt

Quelle Basiswissen für Softwarearchitekten 4. Auflage

## Proxy (R1)

Ein Client muss auf die Operationen einer Instanz einer bestimmten Klasse zugreifen.

Nun kann es passieren, dass der direkte Zugriff auf die Operationen der Klasse nicht möglich, schwierig oder unangebracht ist.

Beispielsweise kann der direkte Zugriff unsicher oder ineffizient sein oder man befindet sich in einem verteilten Umfeld. Hier mag es nicht gewollt sein, dass die physikalische Netzadresse für den direkten Zugriff auf ein verteiltes Objekt im Client hartcodiert ist.

Aber ohne diese Adresse ist ein direkter Zugriff über das Netzwerk nicht möglich. Hier kann das Stellvertreter-Muster helfen. Der Client kommuniziert mit einem Stellvertreter (engl. Proxy) statt mit einer Instanz der eigentlichen Klasse.

Der Proxy bietet dieselbe Schnittstelle an wie die Instanzen der Klasse, die aufgerufen werden sollen. Intern leitet der Proxy den Aufruf an eine Instanz dieser Klasse weiter.



Abbildung Das Proxy Muster

􀁑 Der **Client** stellt das Objekt dar, das durch den Proxy auf das reale Subjekt

zugreift.

􀁑 Der **Proxy** bietet nach außen hin eine zum realen Subjekt identische Schnittstelle.

􀁑 Das **Subjekt** definiert die gemeinsame Schnittstelle von Proxy und realem

Subjekt und ermöglicht die Verwendung von Stellvertretern anstatt realer

Subjekte.

􀁑 Das **reale Subjekt** ist das durch den Proxy repräsentierte Objekt.

## Plugin (R1)

Das Plugin Muster benutzt i.a. das Separated Interface Pattern. Im Separated Interface Pattern definiert die Schnittstelle in einem anderen Paket, in der die Implementationen definiert sind.

Ein Plugin wird i.a. verwendet, in dem man erst zur Konfiguration oder ggf. erst zur Laufzeit das Verhalten bestimmen will. Im Gegensatz zu Konfigurationsdateien, die i.a. nur key value Werte enthalten bestehen PlugIns aus Dateien mit gesamten SourceCode, die das SeparatedInterface erfüllt.



Abbildung Die Plugin Factory erzeugt aus einer Textdatei eine entsprechende Klasse

Ein Plugin wird verwendet, um Code, der erst zur Laufzeit bekannt ist, einbauen zu können.

## Blackboard (R2)

Das Blackboard Muster wird vor allem in Bereichen der Künstlichen Intelligenz verwendet. Es gibt i.a. viele wissensbasierte Systeme (Knowledge Source). Die Ausgangsbasis und die Zwischenergebnisse werden auf einem Blackboard festgehalten.



Es gibt i.a. genau 1 Control Objekt, dass das Blackboard in einer Dauerschleife (loop) überwacht. Meist reagiert es auf Änderungen im Blackboard (nextSource()).

Das Control Objekt triggert die wissensbasierten Systeme, die i.a. unabhängig voneinander sind. Die execCondition entscheidet i.a. ob das wissensbasierte System aktiv wird. Dazu ruft das wissensbasierte System die inspect Methode des Blackboards auf. Innerhalb des execAction werden die Ergebnisse wieder auf das Blackboard zurückgeschrieben (update)

Das Control Objekt triggert die wissensbasierten Systeme, die i.a. unabhängig voneinander sind. Die execCondition, entscheidet i.a. ob das wissensbasierte System aktiv wird.



Das Blackboard selbst, enthält nur die Daten und ist unabhängig vom Control Objekt und den wissensbasierten Systemen.

## Model-View-Controller und Varianten (R2)

## Broker (R2)

Sie können das Broker-Muster verwenden, um die Zusammenarbeit in verteilten Systemen zu koordinieren. Ein Broker verantwortet dabei den Ablauf der Kommunikation verteilter Bausteine sowie die technische Übertragung von Anfragen, Ergebnissen und Ausnahmen

Quelle Software Architektur kompakt

Eine genauere Beschreibung ist in dem Buch Patterm Orientend Software Architecture - A System of Patterns Volume 1 mit u.a. Frank Buschmann als einer der Hauptautoren.

Siehe auch <https://www.grimm-jaud.de/index.php/private-vortraege/27-broker-pattern> als Quelle

Der Unterschied zu RPC, das im nächsten Kapitel beschrieben wird, ist dass die teilnehmenden Komponenten entkoppelt sind. D.h. man entscheidet erst zur Laufzeit, welcher Dienst von welchem Rechner im Netzwerk verwendet wird.



Abbildung Komponenten eines Brokers



Der Server stellt die Funktionalität zur Verfügung, die durch eine Schnittstellen-Definitionssprache spezifiziert wird. Hier muss man zwischen zwei Servertypen unterscheiden:

* Server, die Funktionalität für die Broker Architektur implementieren (Namensdienst )
* Server, die die spezifische Funktionalität zur Verfügung stellen

Anwendungen, die Funktionalität der Server nutzen, indem sie dem Vermittler Anfragen stellen. Die klassische Trennung Client - Server existiert hier nicht, da Server für die Bearbeitung ihrer Anfrage anderere Server nutzen können (müssen).



Ein anderer Name für Client-Side Proxy ist Stub. Der Proxy auf der Server Seite nennt Man auch Skeleton.

Dieser Proxy befindet sich transparent zwischen dem Clienten und dem Vermittler. Seine Aufgabe besteht im Wesentlichen darin, systemspezifsche Funktionalität zu kapseln. Dies kann die Verwaltung des Speichers, der IPC-Mechnismus, das Marshallen oder das Cachen von Daten sein. Oft findet im clientseitigen Proxy eine Abbildung vom Objektmodell des Client zu dem des Vermittlers statt.

 Der Server Proxy erfüllt die ähnlichen Aufgaben wie der Client Proxy. So übersetzt er die Daten in das Objektformat des Servers.



Bringt den Dienstanbieter und den Dienstnutzer zusammen und stellt ihnen einen Kommunikationskanal zur Verfügung. Der Vermittler nutzt zur Auffindung des Service den Namensdienst. Typischerweise besitzt er eine Registratur für die Services und die Mächtigkeit, den Lebenszyklus der Services zu steuern. Serviceanfragen, die er nicht an seine Services delegiert, leitet er gegebenfalls an anderem Broker mittels einer Brücke weiter.



## Remote Procedure Call (R2)

Falls Sie Systeme uber Funktionen anstatt über deren Daten integrieren mochten, stehen Ihnen die *Remote Procedure Calls* als Losungsansatz zur Verfügung. Konzeptionell bedeutet RPC den Aufruf einer Funktion auf einem (möglicherweise entfernt, das heist in einem anderen Prozessraum) ablaufenden Programm. Bild 7.17 zeigt das Vorgehen schematisch. Dort

finden Sie zwei zentrale Begriffe, Stubs und Skeletons: Diese Komponenten wickeln für die

beteiligten Systeme die Kommunikation mit dem entfernten Partnersystem ab.

Jedes beteiligte System verantwortet seine Daten selbst, was eine gute Kapselung bedeutet.

Dz

Abbildung Remote Procedure Call

Stubs sind Proxys (Stellvertreter des Servers) auf der Client Seite. Skeleton sind Proxys (Stellvertreter des Clients) auf der Server Seite. Beide verbergen alle technischen Details der Remote-Kommunikation, beispielsweise Serialisierung und Marshalling/Unmarshalling. Marshalling ist das Umwandeln von strukturierten oder elementaren Daten in ein Format, das die Übermittlung an andere Prozesse oder Programme ermöglicht.

Zu den möglichen Nachteilen von *Remote Procedure Calls* gehören folgende:

* Entfernte Funktionsaufrufe sind in Bezug auf Zeit- und Ressourcenbedarf um mehrere

Größenordnungen teurer als lokale Aufrufe. In der Praxis fuhrt das teilweise zu gravierenden

Performance-Problemen.

* Es bedarf einer komplexen technischen Infrastruktur (Middleware), um entfernte Funktionsaufrufe zu ermöglichen. Technologien wie CORBA und andere sind zwar stabil, performant

und in der Praxis bewahrt, erhöhen jedoch die Komplexität von Systemen (und

Projekten) ganz erheblich.

* Durch die komplexe Hardware- und Softwareinfrastruktur von gekoppelten Systemen

gefährden viele potenzielle Fehlerquellen den ordnungsgemäßen Gesamtablauf.

* Enge Kopplung zwischen den beteiligten Systemen:
* Mögliche Blockierung: Wenn das aufgerufene System nicht oder nur sehr langsam

antwortet, kann es zu Blockierung im Aufrufer kommen, weil der im Regelfall wartet, bis die Antwort vorliegt.

* Eine Schnittstellen Änderung im aufgerufenen System hat möglicherweise Konsequenzen

im Aufrufer. Dadurch besteht das Risiko, dass Änderungen nichtlokale Auswirkungen nach sich ziehen. In der schematischen Abbildung müssen Sie bei Änderungen am System B, stets auch System A testen

Quelle Effektive Softwarearchitekturen 8. Auflage Kapitel 7.3.3

## Messaging (R2), z.B. mit Events und Commands

Quelle Effektive Softwarearchitekturen, 8. Auflage Kapitel 4.5.5

Systeme auf Basis von Ereignissen (Events) bestehen grundsätzlich aus zwei Arten von Bausteinen: Ereignisquellen und -senken (*Event-Sources* und *Event-Sinks*). Quellen und Senken kommunizieren in diesem Stil Uber Ereignisse anstatt im Quellcode verankerte Abhängigkeiten miteinander. Die Kommunikationsbeziehung arbeitet unidirektional von Ereignisquelle zu -senke – diese Beziehungen werden in der Regel erst zur Laufzeit hergestellt. Weil so keine direkten Aufrufe implementiert werden, heist dieser Stil auch *implicit invocation style*.

### Ungepufferte Event-Kommunikation

In der *broadcast*-Variante ereignisbasierter Systeme publiziert eine Event-Source Ereignisse auf einem lokalen Netzwerk. Alle verbundenen Empfänger prüfen jedes Ereignis, ob es für den jeweiligen Empfänger interessant ist: Falls ja, wird das Ereignis bearbeitet, falls nein, wird es ignoriert.



Abbildung Bausteine ungepufferter Event Kommunikation

Alternativ können sich Empfänger von Ereignissen (*Subscriber*) bei den Sendern (*Publisher*) registrieren. Sender übermitteln Ereignisse in diesem Fall direkt an Subscriber.

### Message- oder Event Queue Architekture

Die direkte Verbindung zwischen Sendern und Empfängern von Events wird im Architekturstil

der „Message- oder Event-Queue-Systeme“ um Puffer (Warteschlangen) ergänzt.

###

Abbildung Message Queue Puffer für Nachrichten und Events

Ein Erzeuger von Nachrichten schickt diese an die Queue, aus der Empfänger sie entweder synchron oder asynchron abholen. Diese Struktur wird manchmal als fire-and-forget bezeichnet, weil Erzeuger keine Bestätigung Über Erhalt oder Verarbeitung ihrer Nachrichten erhalten.

Insbesondere können Erzeuger keine Annahmen darüber treffen, ob ihre Nachrichten überhaupt bearbeitet werden beziehungsweise in welcher Reihenfolge. Sind diese Informationen für die Erzeuger wichtig oder müssen die Empfänger gar explizite Antworten auf ihre jeweiligen Nachrichten geben, so bedarf das Message-Queue-Modell einiger Erweiterungen, etwa Über explizite Konversationen oder Korrelations-IDs.

Damit können auch mehrere Nachrichten miteinander in Zusammenhang gebracht werden, so dass der ursprüngliche Erzeuger alle Reaktionen auf eine Nachricht nachvollziehen kann.

### Message Service Architekturen

Wir können die Aufgaben der Warteschlange des vorigen Abschnitts noch um allgemeine

Message-Services erweitern – beispielsweise um folgende:

* Routing von Nachrichten an bestimmte Empfänger, etwa *content-based routing*,
* Ändern der Reihenfolge von Nachrichten – solche mit höherer Priorität werden bevorzugt zugestellt,
* Vorverarbeitung von Nachrichten – etwa Archivierung, Verschlusselung, Protokollierung, statistische oder sonstige Auswertung,
* Zusammenfassung mehrerer Nachrichten oder Aufteilung von Nachrichten,
* Überwachung hinsichtlich bestimmter Qualitätseigenschaften – etwa: Werden die Nachrichten vom Empfänger in der gewünschten Zeit verarbeitet? Ist ein bestimmter Empfänger überlastet und muss eine zusätzliche Instanz gestartet werden?
* Zuverlässige Speicherung von Nachrichten, bis die jeweiligen Empfänger die vollständige Verarbeitung bestätigt haben (*reliable messaging*).

#### Kriterien für die Anwendung

* . Erzeuger (Sender) von Nachrichten benötigt für die weitere Verarbeitung keine synchrone Antwort (fire-and-forget).
* Integrationsszenarien: Es müssen mehrere unterschiedliche Systeme zusammenarbeiten.
* Sender erzeugt Nachrichten erheblich schneller als sie von Empfängern

#### Vorteile

* Sender und Empfänger von Nachrichten können in völlig unterschiedlichen Technologien und Programmiersprachen erstellt werden.
* Message-Queues, insbesondere solche mit zuverlässiger Zustellung (*reliable messaging*), können die Verfügbarkeit und Robustheit von Systemen erheblich steigern. Aus diesem Grund werden MQ-Systeme insbesondere im Bereich Finanz- und Kontodaten häufig eingesetzt.

#### Nachteile

* Message-Queues (ob kommerziell oder Open-Source) sind in sich komplexe Systeme mit teilweise hohem Einführung- und Administrationsaufwand.
* Asynchrone und nachrichtenbasierte Programmierung ist signifikant aufwendiger als einfacher call-and-return-Stil. Fehlersuche in asynchronen Systemen kann aufwendig sein.