# Das Rad nicht immer neu erfinden –

## Architekturmuster im embedded Umfeld einsetzen

Frank Listing
f.listing@microconsult.com

MICRO CONSULT

Was sind Muster (Patterns)?

Warum sollten Muster eingesetzt werden?

Wie werden Architekturmuster eingeteilt?

Vorstellung einer Auswahl von Architekturmustern.

# Entwurfsmuster (Patterns)

Entwurfsmuster sind Schablonen, bewährte Lösungsansätze für immer wiederkehrende Probleme in der Softwareentwicklung.

In den Mustern wird auf allgemeine Art die Lösung eines speziellen Problems beschrieben.

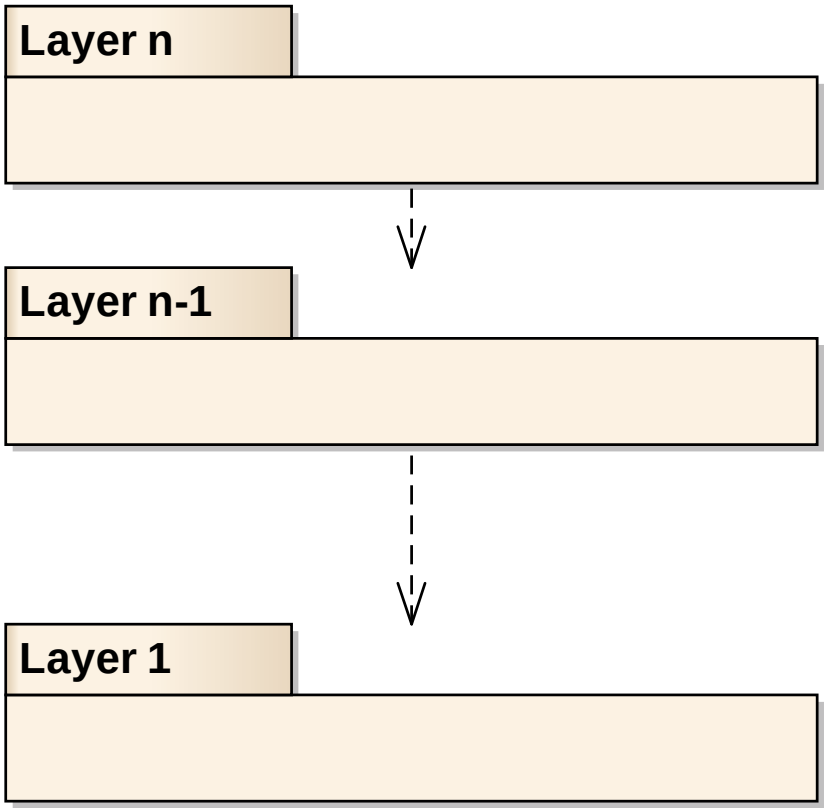
Entwurfsmuster sind unabhängig von einer speziellen Programmiersprache.

Architekturmuster setzen eine Ebene höher an und beschreiben Lösungsansätze für den grundlegenden Aufbau einer Applikation und die Interaktion zwischen den Komponenten.

## Beispiel Schichtenarchitektur

Die Schichtenarchitektur unterteilt die Applikation in Schichten unterschiedlicher (aufsteigender) Abstraktion.

Die obere Schicht ist von der unteren abhängig und nicht umgekehrt.



# Warum Patterns?

Beschleunigung des Architekturentwurfs durch die Nutzung vorhandener Lösungsansätze.

Erleichterung bei der Kommunikation mit anderen am Entwurf Beteiligten.

Gegenüberstellung von Entwurfsvarianten – Vor- und Nachteile der einzelnen Muster sind dokumentiert.

Sicherheit im Entwurf durch die Nutzung bewährter Lösungen.

Die Einteilung der Architekturmuster erfolgt im Allgemeinen in vier Kategorien:

**Chaos zu Struktur (engl. Mud-to-structure)**

- Organisation der Komponenten und Objekte eines Softwaresystems
- Aufteilung der Funktionalität des Gesamtsystems in Komponenten/ Subsysteme

**Verteilte Systeme**

- Verwendung von verteilten Ressourcen und Diensten in einem Netzwerk

Interaktive Systeme

- Einbindung einer Benutzerschnittstelle in ein System

**Adaptive Systeme**

- Erweiterbarkeit und Anpassungsfähigkeit von Systemen

# Chaos zu Struktur (engl. Mud-to-structure)

**Schichtenarchitektur**

- Aufteilung eines Softwaresystems in aufeinander aufbauende Schichten

**Pipes und Filter**

- Das System wird in unabhängige Einheiten (Filter) strukturiert
- Die Filter werden durch Pipes verbunden

**Schwarzes Brett (Blackboard)**

- Ein Schwarzes Brett dient dem zentralen Datenaustausch zwischen Teilprozessen
- Die einzelnen Prozesse ändern die Daten bzw. werden über Änderungen informiert

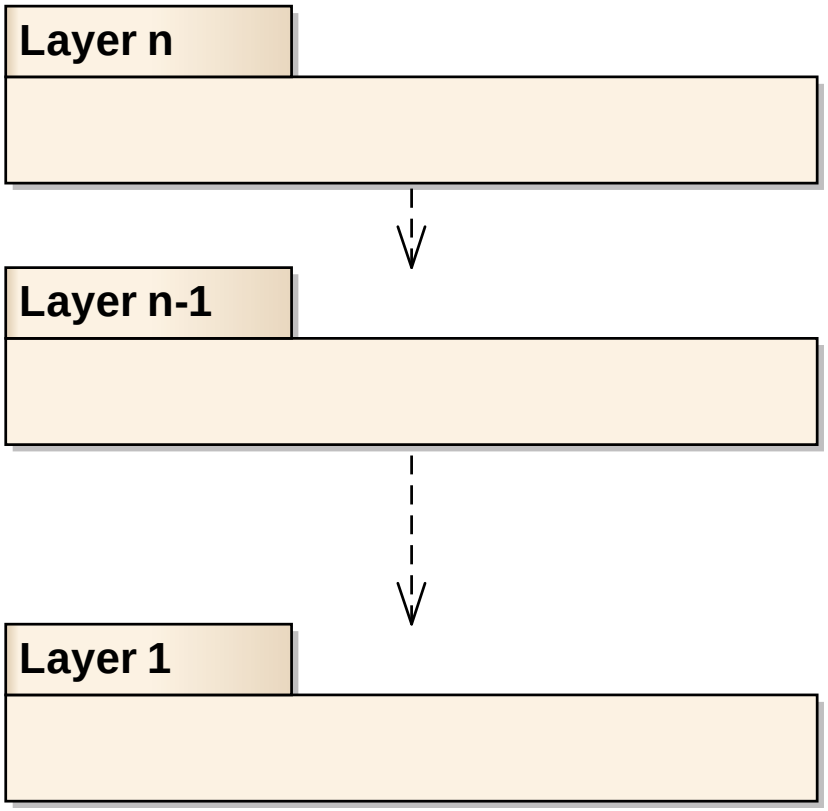
Domain-Driven Design

- Anwendungsdomänen-getriebene Herangehensweise an Architektur und Design

Die Schichtenarchitektur unterteilt die Applikation in Schichten unterschiedlicher (aufsteigender) Abstraktion.

Die obere Schicht ist von der unteren abhängig und nicht umgekehrt.

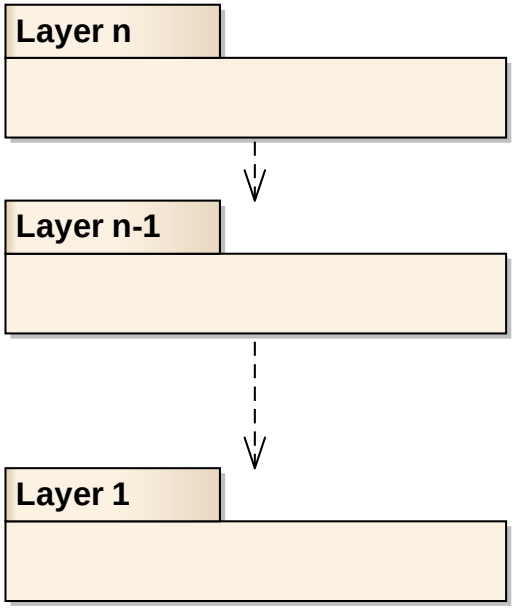
Datenflüsse von unten nach oben werden durch Callback-Mechanismen realisiert. Dies vermeidet zirkuläre Abhängigkeiten.



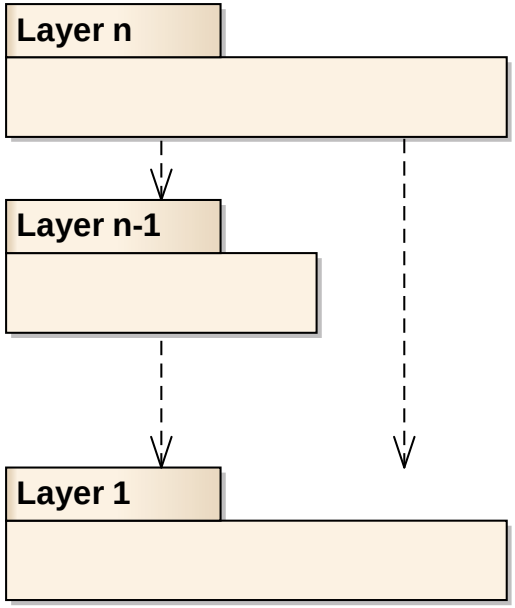
Strikte Schichtenarchitektur – Die obere Schicht greift nur auf die direkt darunterliegende zu.

Nicht strikte Schichtenarchitektur – Es wird auch mal eine Schicht übersprungen.

Strikte Schichtenarchitektur




Nicht strikte Schichtenarchitektur



Vorteile

- Reduzierung von Abhängigkeiten
- Definierte Schnittstellen zwischen den Schichten
- Änderungen wirken sich nur selten auf andere Schichten aus
- Wiederverwendung und Austausch einer Schicht ist möglich
- Erhöhung der Betriebssicherheit (z.B. durch den Einbau von zusätzlichen Schichten zur Überwachung)

Nachteile

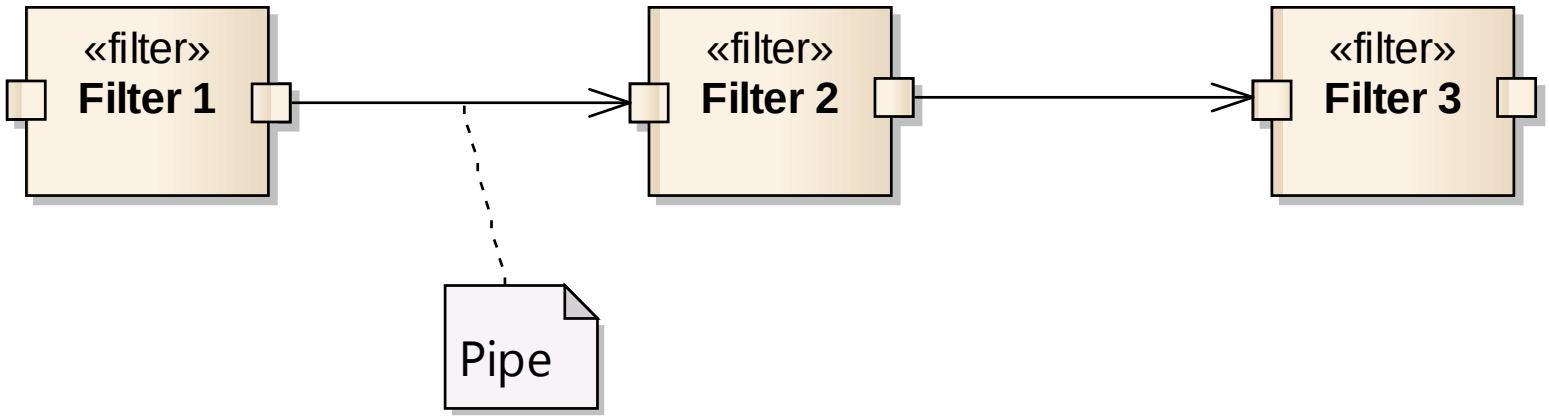
- Eventuell Effizienzverringerung durch Datentransformationen beim Durchreichen in höhere Schichten

Die Aufbereitung von Daten wird in einzelne Teilschritte zerlegt (Filter).

Diese Filter werden über Pipes miteinander verknüpft.

Eine Pipe dient nur dem Datentransport (evtl. mit Zwischenpufferung), ein Filter bereitet Daten auf und übergibt sie einer Pipe.

Ideal für Systeme, die Datenströme verarbeiten.



Vorteile

- Hohe Flexibilität – Filter können ohne großen Aufwand ausgetauscht werden
- Wiederverwendung einzelner Filter
- Bei parallelen Verarbeitungsketten gut auf mehrere Prozessoren verteilbar

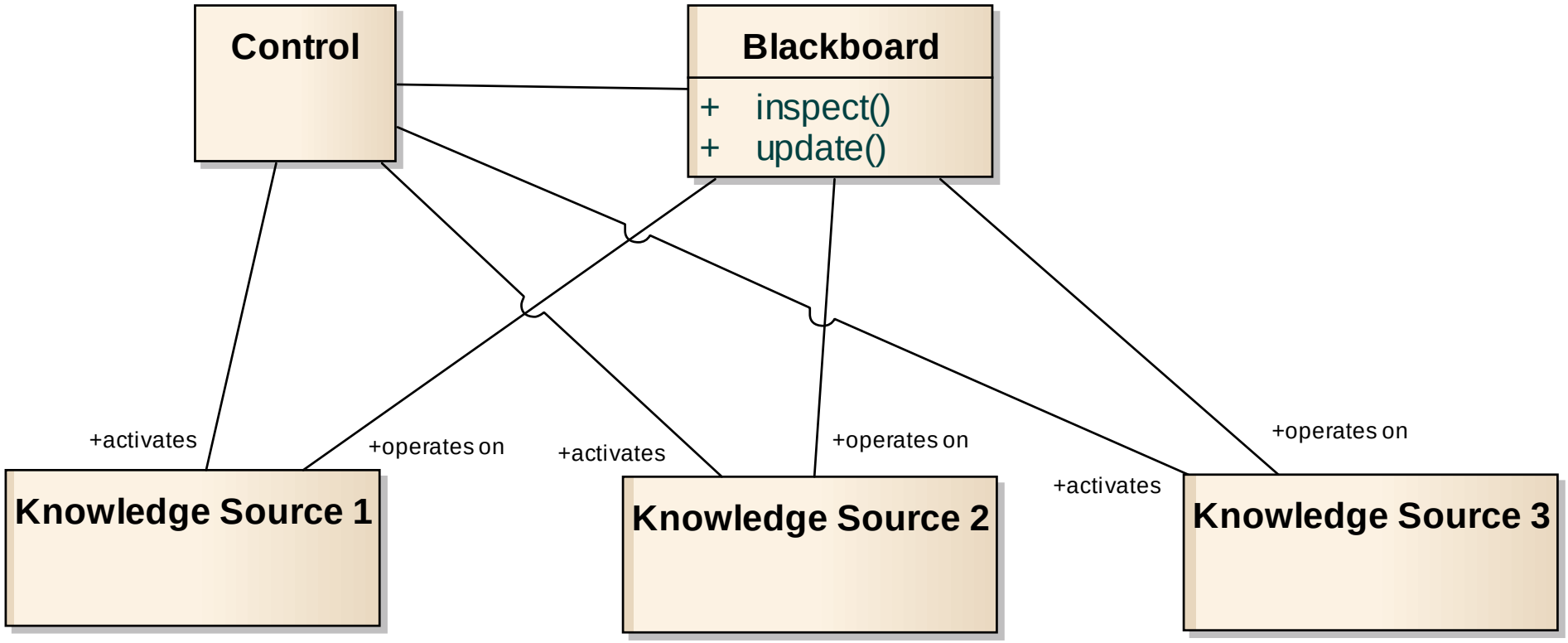
Nachteile

- Eventuell Effizienzverluste durch Datentransformationen in das Datenformat der Pipe
- Abhängigkeit der gesamten Filterkette vom schwächsten Glied (Geschwindigkeit, Stabilität)

Zentrale Austauschplattform für gemeinsame Daten

Einheitliche Datenablage ohne direktem Datenzugriff.

- versenden von Änderungsmitteilungen an Datennutzer
- Validierung von Werten



Vorteile

- Einheitlicher Mechanismus für den Zugriff auf zentrale Daten
- Fehlerhafte Zugriffe auf die Daten sind einfach zu erkennen
- Keine direkten, unkontrollierten Zugriffe auf die Daten
- Änderungen sind einfacher durchzuführen

Nachteile

- Je nach Komfortgrad der Implementierung dauert der Zugriff auf ein Datum länger

# Verteilte Systeme

## Client-Server

- Klare Aufgabenverteilung in Client und Server
- Ein Client nutzt einen Dienst des Servers
- Der Server bietet die Dienste an

## Broker

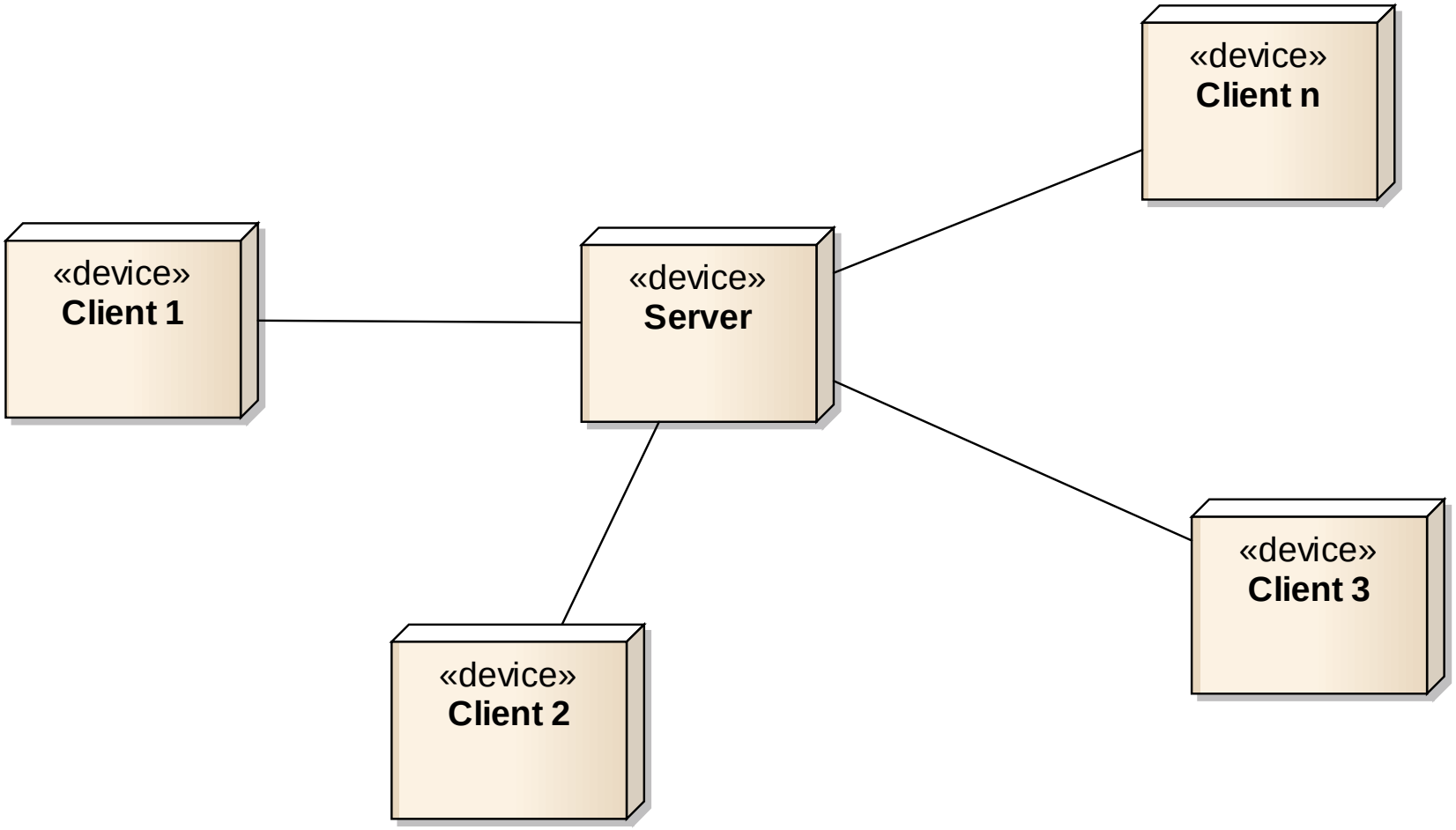
- Dienste im Netzwerk werden über eine zentrale Registrierung, dem Broker, angeboten
- Ein Client nutzt den Broker als zentrale Anlaufstelle, um einen passenden Dienst zu finden

## Peer-to-Peer

- Alle Computer im Netz sind gleichberechtigt
- Jeder Computer kann Dienste nutzen und anbieten

Ein Server bietet einen oder mehrere Dienste in einem Netzwerk an.

Clients können sich zum Server verbinden und diese Dienste in Anspruch nehmen.



Varianten der Client-Server-Architektur

Fat Client

- Hoher Anteil der Datenverarbeitung auf dem Client
- Geringe Serverlast
- Komplexe Client-Software

Thin Client

- Hohe Beanspruchung des Servers
- Hohe Netzwerklast
- Einfache Client-Software

Ultra Thin Client

- Extremfall des Thin Clients, z.B. nur Webbrowser

## Vorteile

- Zentralisierung von Aufgaben
- Einfache Wartung der zentralen Software
- Geringe Hardwarekosten bei den Clients

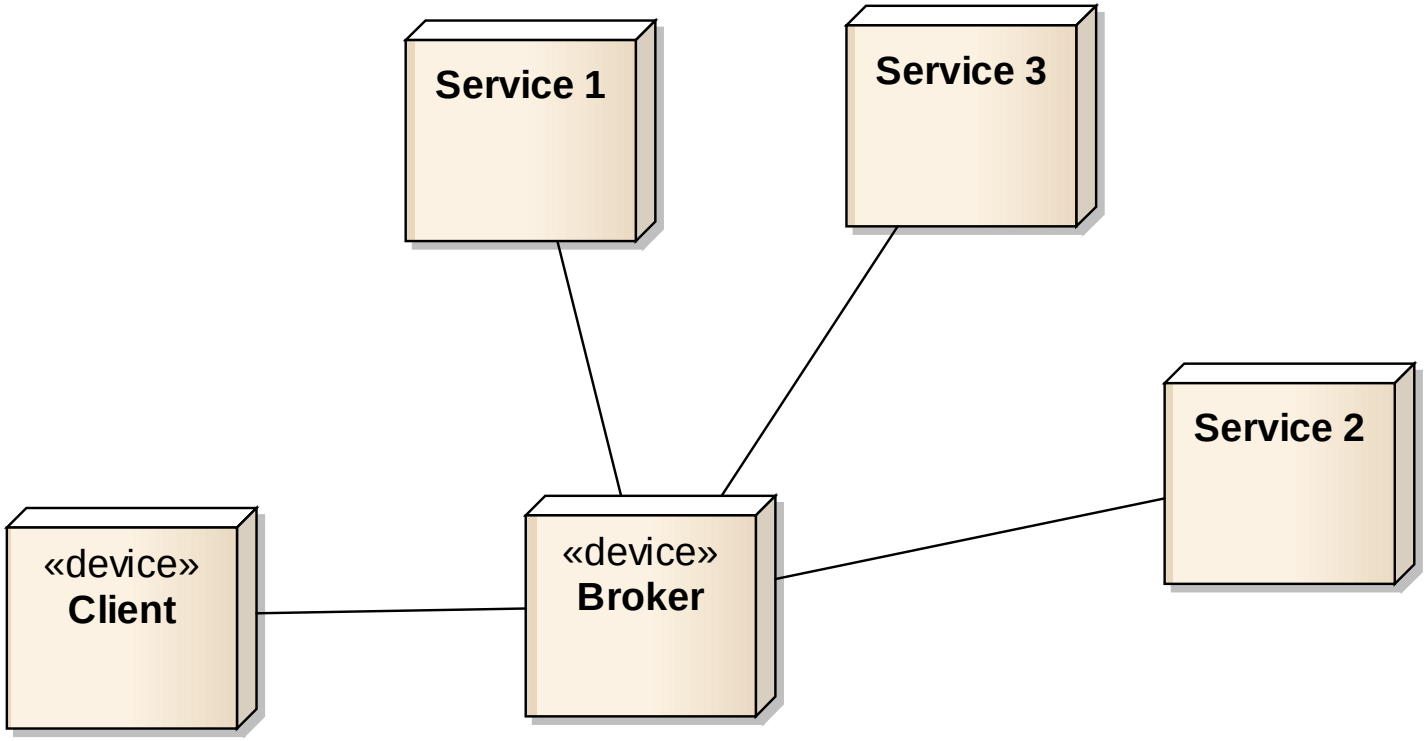
## Nachteile

- Stillstand bei Netzwerk- bzw. Serverausfall
- Netzwerk bzw. Server als Flaschenhals

Mehrere Server bieten Dienste im Netzwerk an.

Clients wollen einen oder mehrere Dienste nutzen, kennen aber die Anbieter nicht. Lediglich der Broker ist dem Client bekannt.

Der Broker vermittelt zwischen Client und Server, um für die zu lösende Aufgabe den richtigen Dienst zu finden.



## Vorteile

- Der Client hat nur einen Ansprechpartner und muss nicht jeden Server kennen
- Einfache Austauschbarkeit von Servern, Änderungen müssen nur dem Broker bekannt gegeben werden

## Nachteile

- Laufzeitnachteile durch zwischengeschalteten Broker
- Hohe Netzwerklast
- Ausfall des Brokers legt gesamtes System lahm
- Komplizierter Test

# Adaptive Systeme

## Mikrokernel

- Systemkern mit Minimalfunktionalität
- Weitergehende Funktionalität wird aus dem Mikrokernel heraus entwickelt

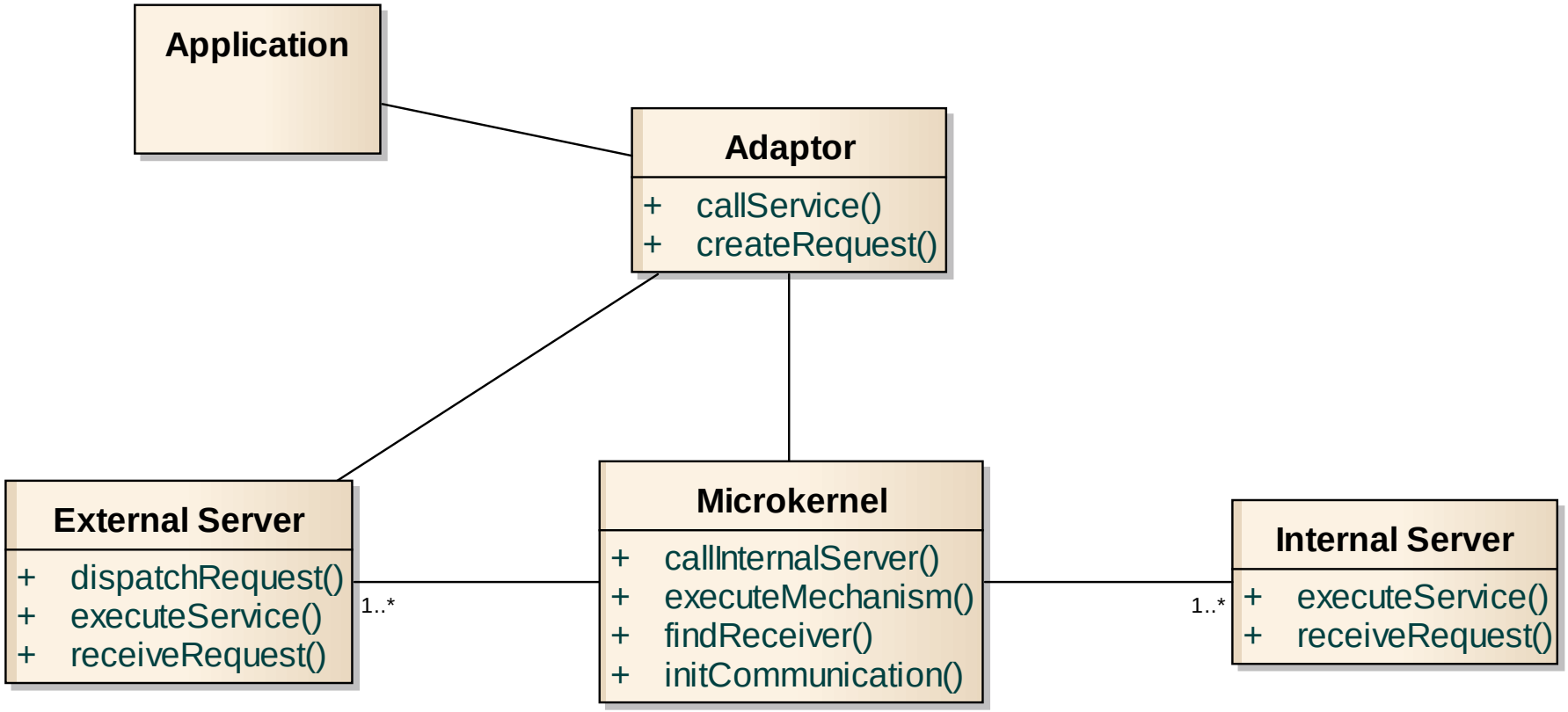
## Dependency Injection

- Reduzierung von Abhängigkeiten beim Kompilieren
- Objekterzeugung und -verknüpfung erfolgt durch zentrale Komponente zur Laufzeit

Der Microkernel stellt nur Kernfunktionen zur Verfügung.

Die Applikation arbeitet nur mit den externen Servern, die aufbauend auf dem Microkernel die Systemfunktionalität bereitstellen.

Externe Server laufen in einem anderen Prozess als der Mikrokernel.



## Vorteile

- Gute Wartbarkeit
- Erweiterbar ohne den Kern zu ändern
- Einfacher auf andere Hardware zu portieren

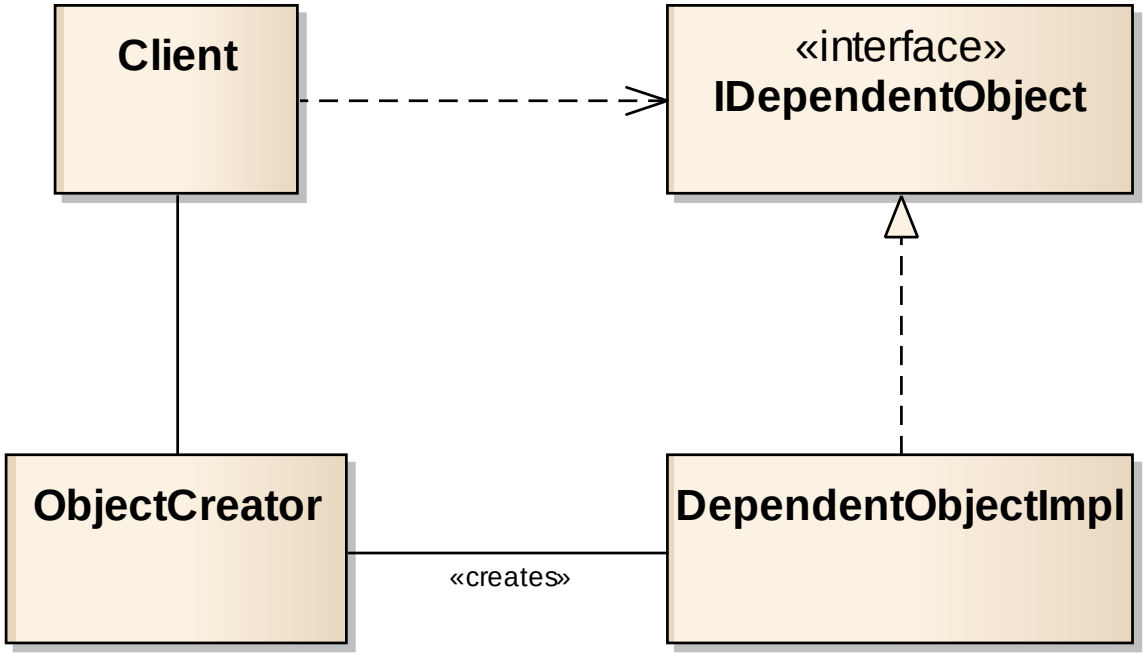
## Nachteile

- Durch Rückführung aller Funktionen auf die Funktionalität des Microkernels, kann dieser zum Flaschenhals werden
- Geschwindigkeitsverlust durch zwischengelagerte externe Server

Der Client erzeugt abhängige Objekte nicht selbst. Das wird von einer eigens dafür angelegten Klasse erledigt.

Die einzige Abhängigkeit des Clients besteht zum Interface.

Die konkrete Implementierung kennt nur noch die Erzeugerklasse und ist damit einfach austauschbar.



## Vorteile

- Einfache Austauschbarkeit der konkreten Implementierung
- Weniger Abhängigkeiten
- Bessere Testbarkeit der Klassen
- Zentralisierte Objekterzeugung

## Nachteile

- Zusätzliche Erzeugerklasse
- Geringere Übersichtlichkeit

# Safety Patterns

## Homogeneous Redundancy Pattern

- Das System wird verdoppelt.
- Fällt eine Hälfte aus, arbeitet die andere weiter.

## Triple Modular Redundancy Pattern

- Das System wird verdreifacht.
- Die Ergebnisse aller Komponenten werden von einer Entscheidungskomponente ausgewertet, weicht das Ergebnis eines Teilsystems ab, wird es verworfen.

## Heterogeneous Redundancy Pattern

- Das System wird mindestens verdoppelt.
- Alle Teilsysteme werden von unterschiedlichen Entwicklern programmiert.
- Im Gegensatz zu den homogenen Systemen werden auch systematische Fehler gefunden.

# Multi-Core-/Multi-Processor Patterns

## Task Parallelism Pattern

- Eine Applikation wird in eine Menge von unabhängigen Tasks zerlegt, die gleichzeitig ausgeführt werden können.

## Event-Based Coordination Pattern

- Eine Applikation wird in eine Menge von abhängigen Tasks zerlegt. Die Synchronisation erfolgt über Events. Damit kann ein Teil der Aufgaben gleichzeitig ausgeführt werden.

## Divide and Conquer Pattern

- Eine Aufgabe wird in parallel ausführbare Teilaufgaben zerlegt. Die Ergebnisse werden nach der Verarbeitung zum Gesamtergebnis zusammengesetzt.

## Geometric Decomposition Pattern

- Eine große Datenstruktur wird in kleinere separat bearbeitbare Abschnitte zerteilt. Diese werden parallel bearbeitet.

# Real-Time Patterns

## Message Queuing Pattern

- Aufgaben, die in einer separaten Task abgearbeitet werden sollen, werden als Nachricht in die Queue dieser Task geschrieben.
- Wenn die Task läuft, liest sie die Nachrichten in der Queue und arbeitet sie ab.
- Da die Abarbeitung seriell erfolgt, gibt es keine Ressourcenkonflikte.

## Cyclic Execution Pattern

- Alle Tasks stehen in einer Liste und werden nacheinander immer wieder abgearbeitet.
- Einfacher Scheduling-Mechanismus.

## Pooled Allocation Pattern

- Anstatt dynamischer Speicheranforderung
- Es wird eine Anzahl von Objekten vorgehalten. Wird von einer Task ein Objekt benötigt, wird es initialisiert und zugewiesen. Hinterher kommt es wieder in den Pool.

→

Architekturmuster sind eine solide Basis für die Entwicklung einer Applikation.

Sie helfen den Entwicklungsaufwand zu verringern.

Es wird auf bewährte Lösungen zurückgegriffen.