Muster

Christian Silberbauer

Einführung

- Definition:

  > Ein Pattern (Muster) ist eine bewährte Lösung eines Problems in einem bestimmten Kontext.

- Patterns wurden erstmals von Christopher Alexander in den 1970er Jahren für das Architekturumfeld definiert.

- Entwurfsmuster in der Softwaretechnik wurden erstmals 1995 im Buch *Design Patterns* von Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides, auch als *Gang-of-Four* bzw. *GoF* bekannt, beschrieben.

- Das Wissen über Muster erleichtert den Softwareentwurf und die Kommunikation dessen. Es erhöht das Abstraktionsniveau in der Programmierung.

Architektur- und Entwurfsmuster

- Architekturmuster:

> An architectural pattern expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their repsonsibilities, and includes rules and guidelines for organizing the relationships between them.

- Entwurfsmuster:

> A design pattern provides a scheme for refining the subsystems or components of a software system, or the relationships between them. It describes a commonly-recurring structure of communication components that solves a general design problem within a particular context.

- Mustersammlung: http://www.hillside.net/patterns
- Quelle: (Buschmann, 1996), S. 12

Architektur- und Entwurfsmuster

Entwurfsmuster stellen **feinkörnige** Muster dar, während Architekturmuster **grobkörnige** Muster sind.

- Quelle: (Goll, 2014), S. 69

Beschreibung von Entwurfsmustern

- Nach GoF:
  - Name and Classification
  - Intent
  - Also Known As
  - Motivation
  - Applicability
  - Structure
  - Participants
  - Collaborations
  - Consequences
  - Implementation
  - Sample Code
  - Known Uses
  - Related Patterns
- Nach Buschmann:
  - Name and Classification
  - Also Known As
  - Context
  - Problem
  - Solution
  - Structure
  - Dynamics
  - Implementation
  - Example Resolved
  - Variants
  - Known Uses
  - Consequences
  - See Also

Entwurfsmuster

- Creational-Patterns
  - Abstract-Factory
  - Builder
  - Factory-Method
  - Prototype
  - Singleton
  - Object-Pool
- Structural-Patterns
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Facade
  - Flyweight
  - Proxy
- Behavioral-Patterns
  - Chain-of-Responsibility
  - Command
  - Interpreter
  - Iterator
  - Mediator
  - Memento
  - Observer
  - State
  - Strategy
  - Template-Method
  - Visitor

23 „klassische“ Muster aus GoF
+ Object-Pool aus Goll

Entwurfsmuster

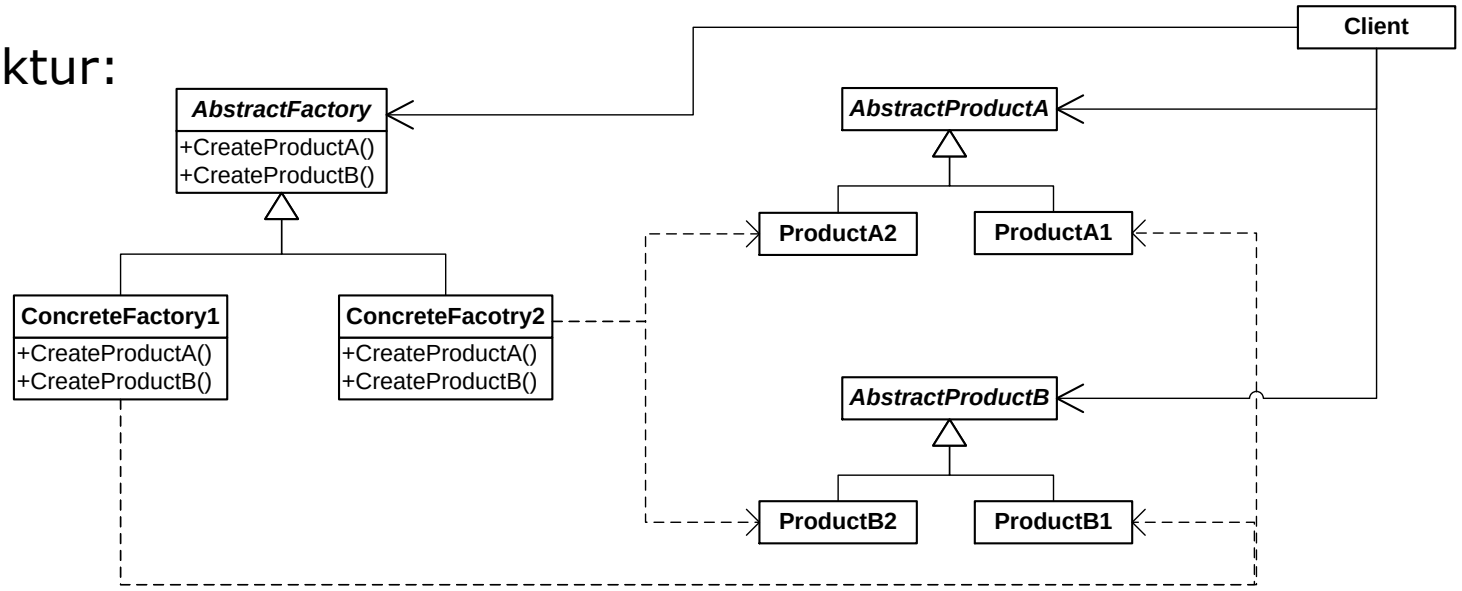
- **Creational-Patterns**
  - **Abstract-Factory**
  - **Builder**
  - **Factory-Method**
  - **Prototype**
  - **Singleton**
  - **Object-Pool**
- Structural-Patterns
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Facade
  - Flyweight
  - Proxy
- Behavioral-Patterns
  - Chain-of-Responsibility
  - Command
  - Interpreter
  - Iterator
  - Mediator
  - Memento
  - Observer
  - State
  - Strategy
  - Template-Method
  - Visitor

Abstract-Factory-Muster

- Zweck:

  Ermöglicht eine Schnittstelle, die zum Erzeugen von verwandten Objekten oder Methoden dient, ohne ihre konkreten Klassen zu kennen.

- Struktur:



Abstract-Factory-Muster

- Beispiel:
  - Dabei stehen `ProductA` und `ProductB` z.B. für `Button` oder `TextField`.
  - Produktfamilien (`1` oder `2`) stehen z.B. für „Windows-Syle“ oder „Linux-Style“.
  - „Consider a user interface toolkit that supports multiple look-and-feel standards, such as Motif and Presentation Manager…“

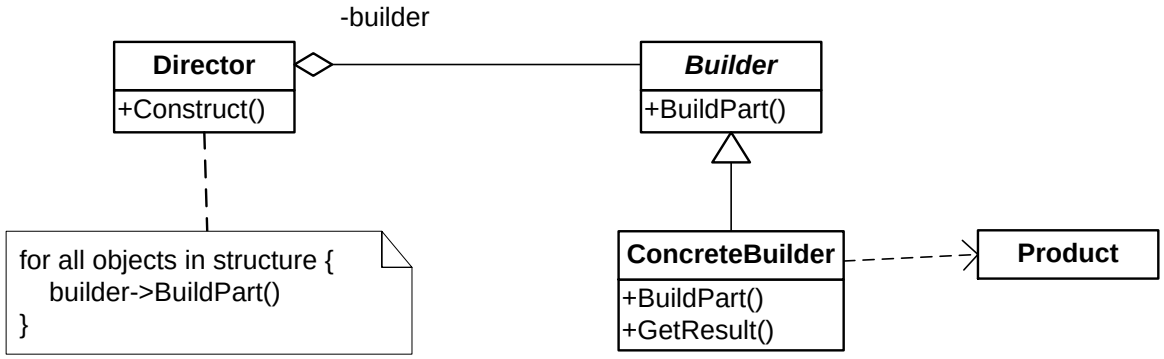
- Konsequenzen:
  - Es isoliert konkrete Klassen.
  - Es macht den Austausch von Produktfamilien einfach.
  - Es fördert Konsistenz unter Produkten.
  - Die Unterstützung neuer Arten von Produkten ist schwierig.

Builder-Muster

- Zweck:

> Trennt die Konstruktion eines komplexen Objekts von seiner Repräsentation, sodass derselbe Konstruktionsprozess verschiedene Repräsentationen erzeugen kann.

- Struktur:

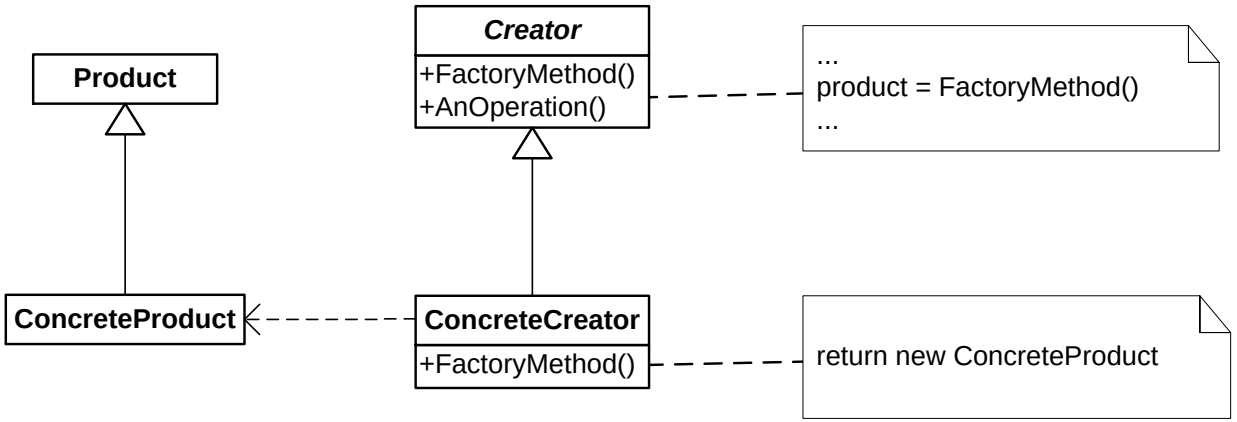


Factory-Method-Muster

- Zweck:

> Definiert ein Interface, um Objekte zu erzeugen, aber lässt abgeleiteten Klassen entscheiden, welche Klasse instanziiert wird.

- Struktur:

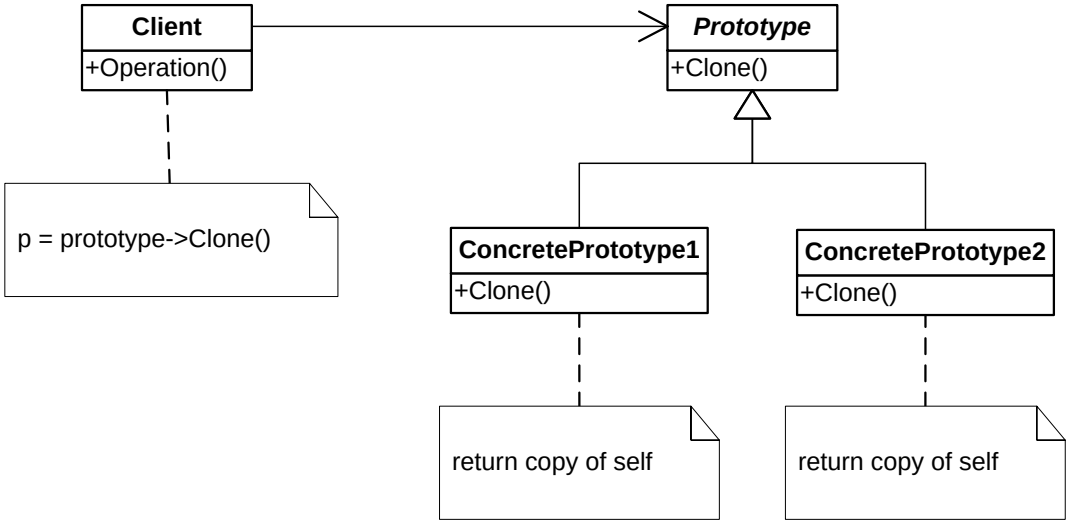


Prototype-Muster

- Zweck:

> Spezifiziert die Art eines Objekts bei der Erzeugung unter Verwendung einer prototypischen Instanz und erstellt neue Objekte durch Kopieren dieses Prototyps.

- Struktur:

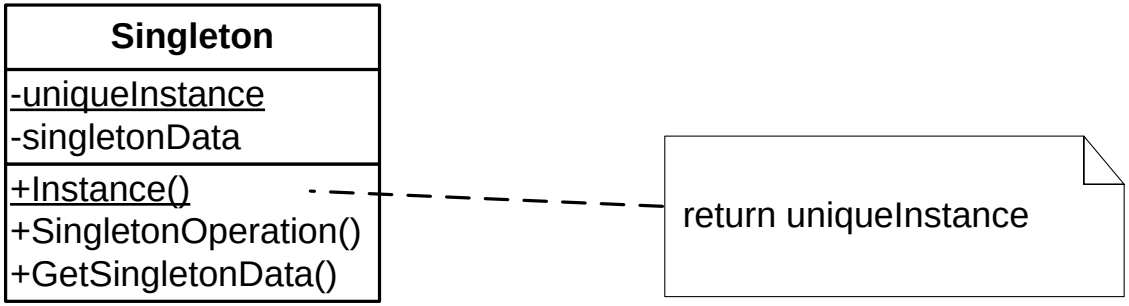


Singleton-Muster

- Zweck:

  Stellt sicher, dass es von einer Klasse nur eine Instanz gibt und bietet einen globalen Zugriff auf dieses Objekt.

- Struktur:

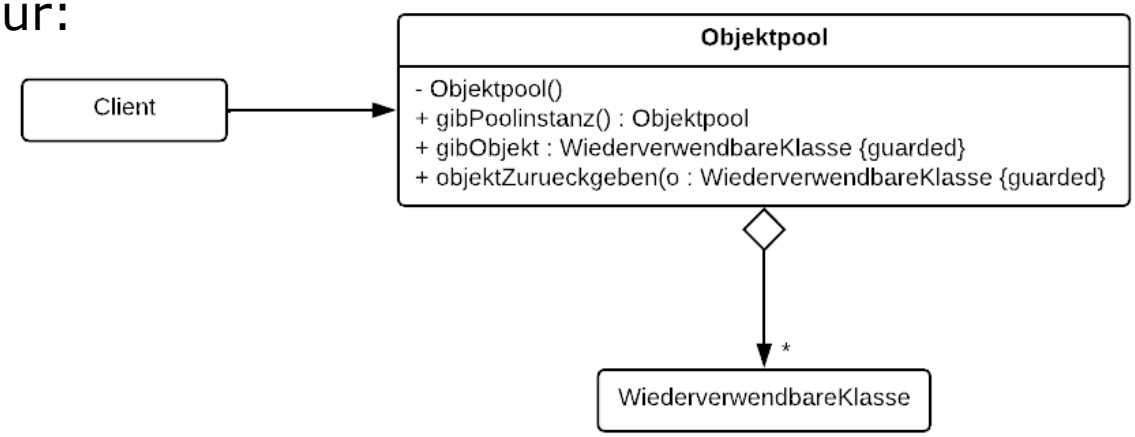


Object-Pool-Muster

- Zweck:

  Ermöglicht es, Objekte aus einem Pool zur Verfügung zu stellen und wiederzuverwenden.

- Struktur:



- Beispiele: Connection-, Thread-, Service-Pool

Entwurfsmuster

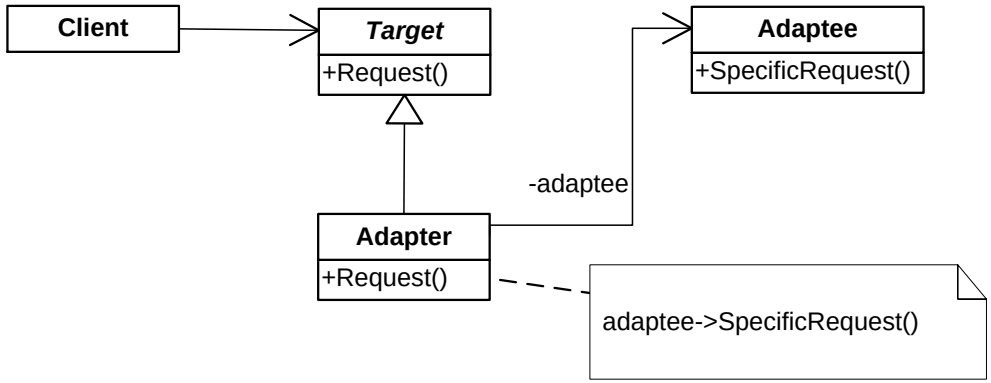
- Creational-Patterns
  - Abstract-Factory
  - Builder
  - Factory-Method
  - Prototype
  - Singleton
  - Object-Pool
- **Structural-Patterns**
  - **Adapter**
  - **Bridge**
  - **Composite**
  - **Decorator**
  - **Facade**
  - **Flyweight**
  - **Proxy**
- Behavioral-Patterns
  - Chain-of-Responsibility
  - Command
  - Interpreter
  - Iterator
  - Mediator
  - Memento
  - Observer
  - State
  - Strategy
  - Template-Method
  - Visitor

Adapter-Muster

- Zweck:

> Konvertiert das Interface einer Klasse in ein anderes Interface, das Clients erwarten. Adapter ermöglichen die Zusammenarbeit von Klassen, die dies sonst wegen inkompatibler Schnittstellen nicht könnten.

- Struktur (Objektadapter):

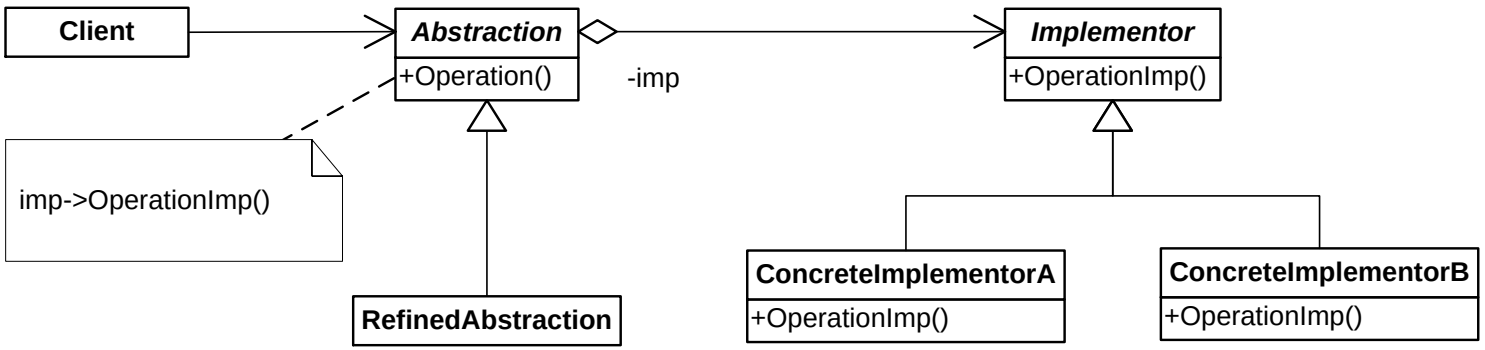


Bridge-Muster

- Zweck:

> Entkoppelt eine Abstraktion von ihrer Implementierung, so dass beide unabhängig voneinander variiert werden können.

- Struktur:

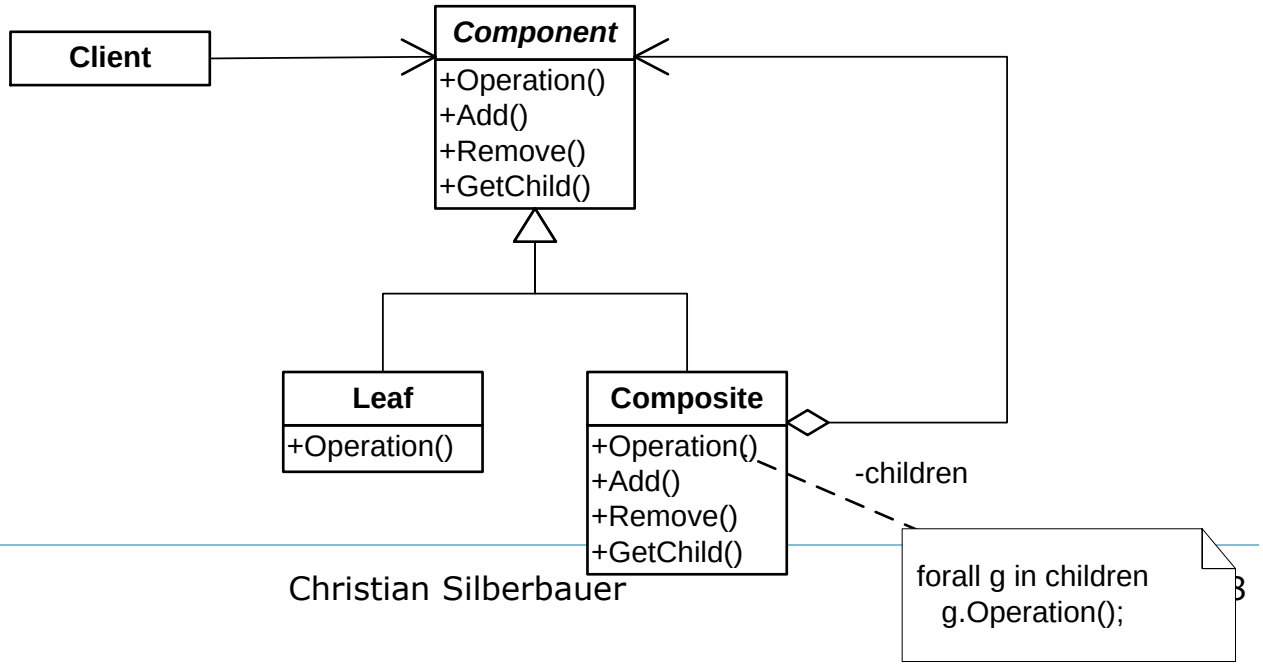


Composite-Muster

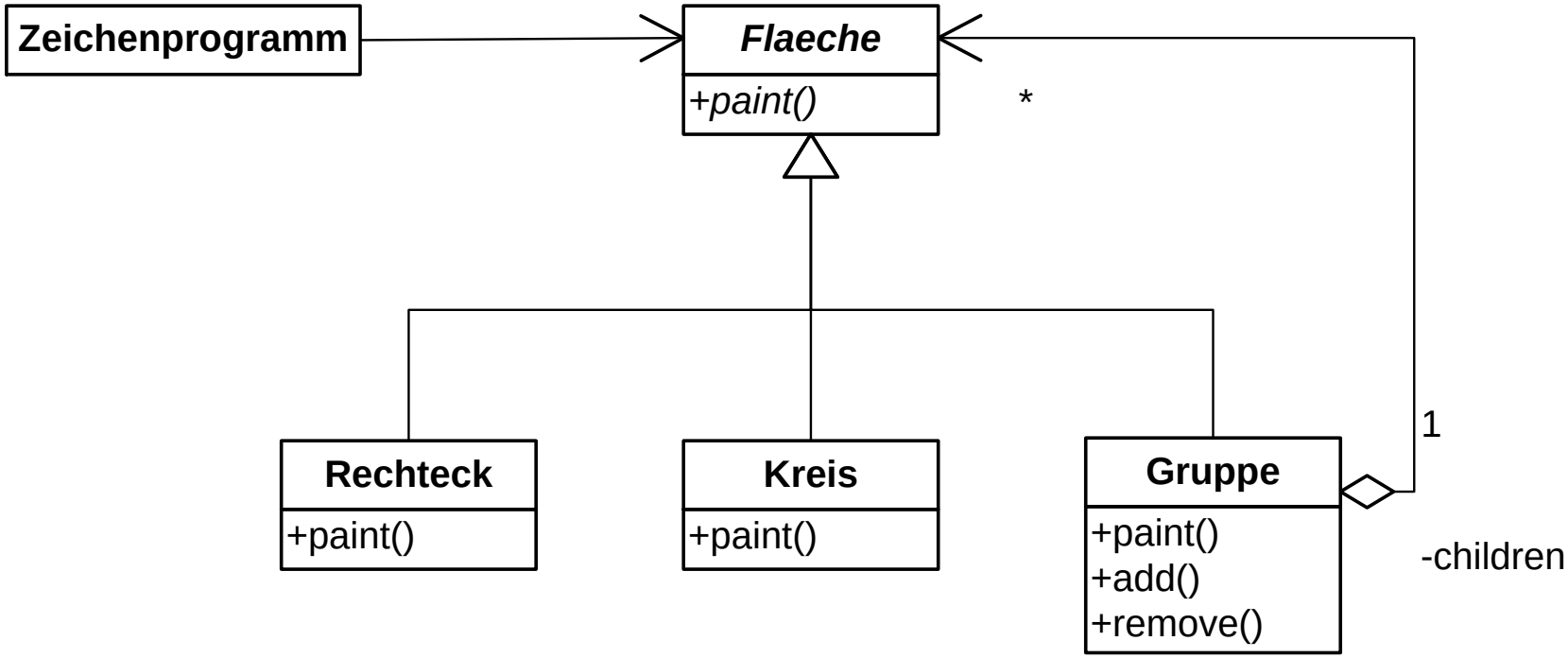
- Zweck:

Ermöglicht es, Objekte zu einer Baumstruktur zusammenzusetzen und Teil/Ganzes-Hierarchien auszudrücken. Composite erlaubt den Clients, individuelle Objekte und Zusammensetzungen von Objekten auf gleiche Weise zu behandeln.

- Struktur:



Beispiel `Zeichenprogramm`

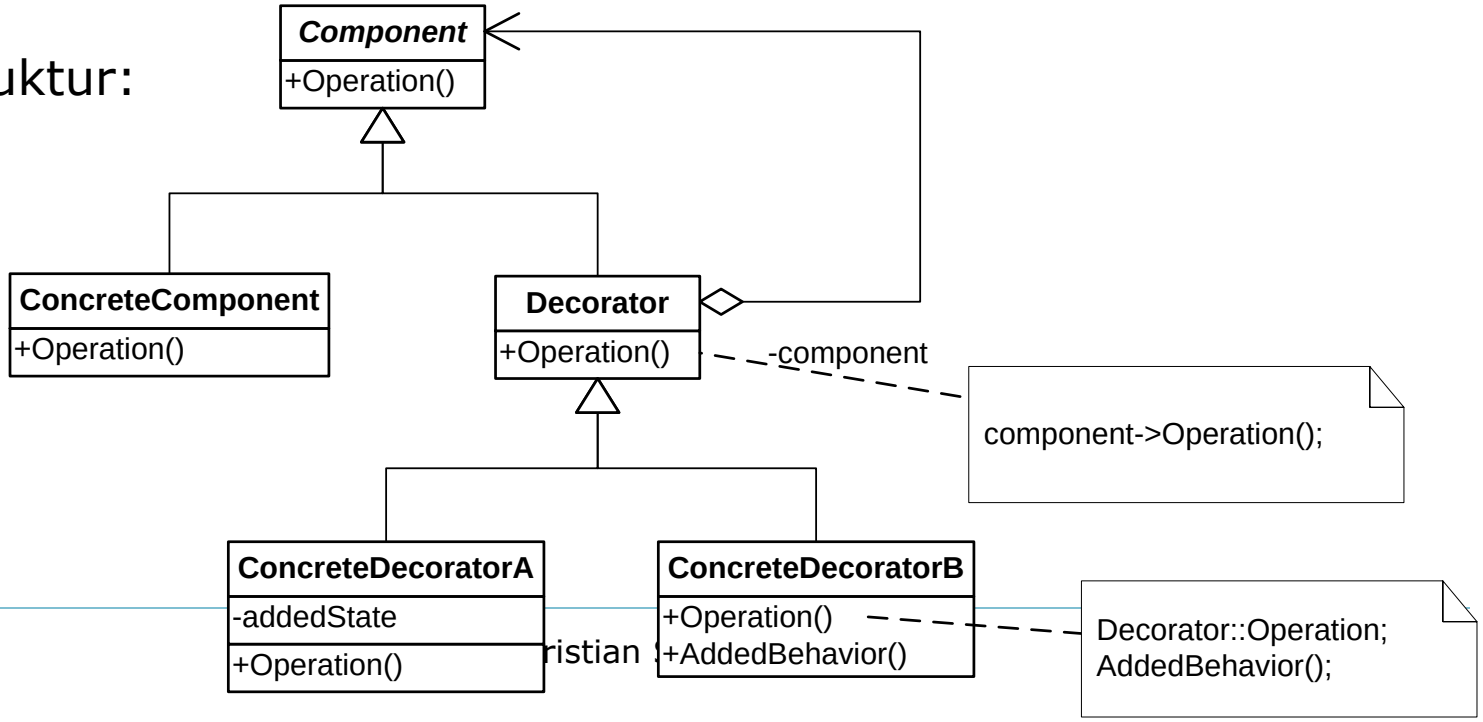


Decorator-Muster

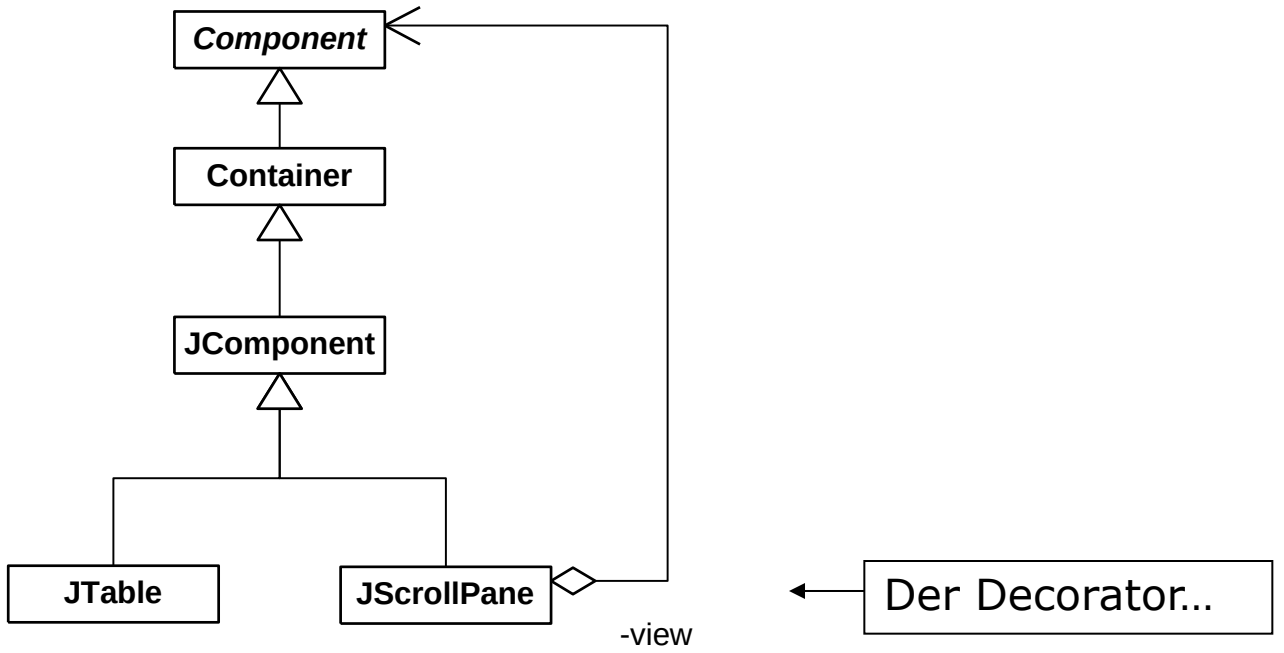
- Zweck:

> Fügt einem Objekt dynamisch zusätzliche Verantwortlichkeiten hinzu. Decorators bieten eine flexible Alternative zur Vererbung zum Zweck der Erweiterung der Funktionalität.

- Struktur:

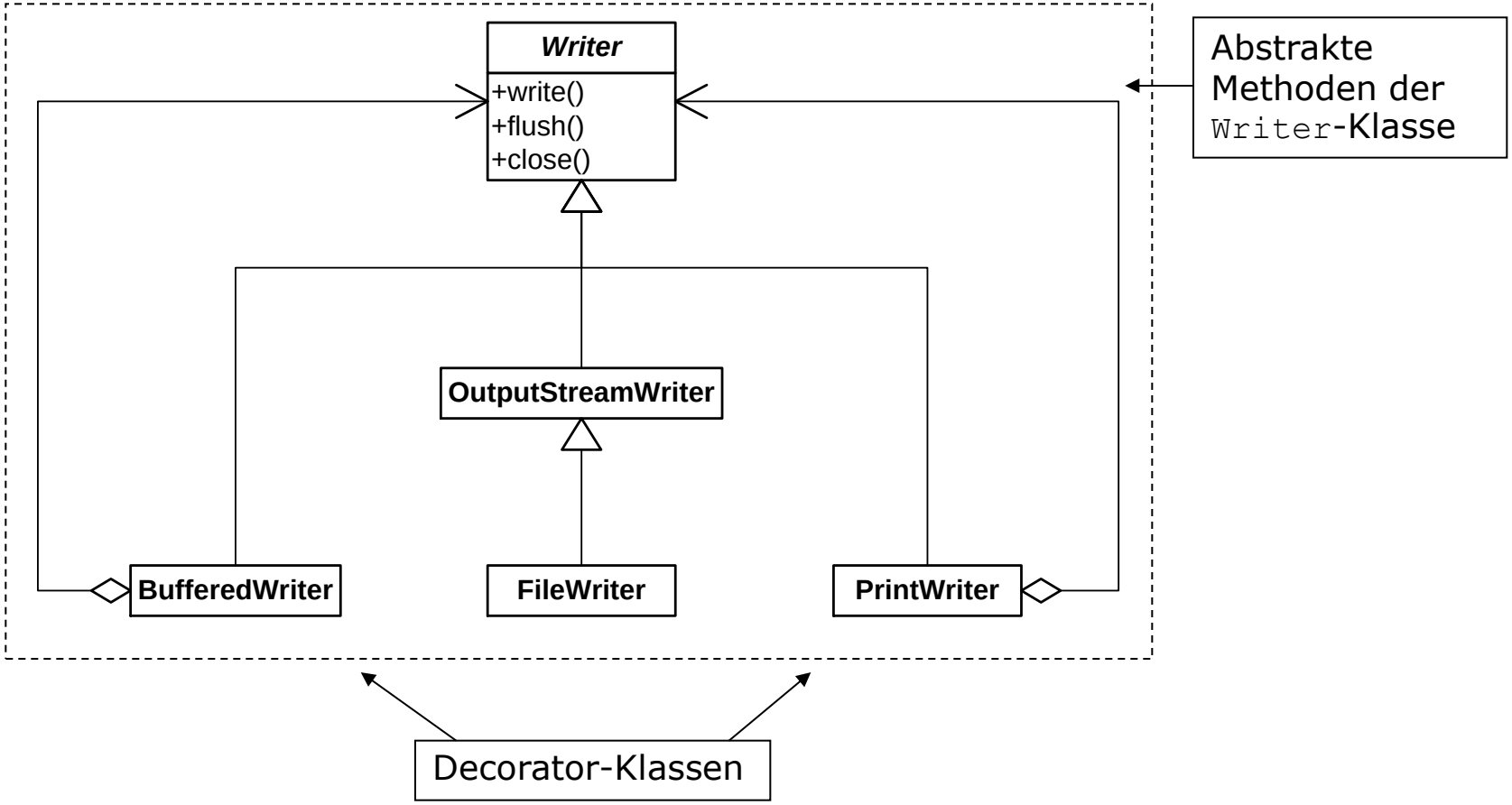


Decorator-Muster



- Ein detaillierteres Beispiel zum Decorator-Pattern folgt beim Thema *Dateiverarbeitung* im Kapitel *Persistenz*

Beispiel `java.io`



Beispiel `java.io`

- `BufferedWriter` und `PrintWriter` dekorieren `Writer`-Klassen und erweitern somit ihre Funktionalität um Pufferung bzw. Formatierungsmöglichkeiten.
- Zu diesem Zweck erben auch Decorators von der Klasse `Writer`.

- Sie besitzen zudem eine Oberklassen-Referenz auf den zu dekorierenden `Writer`.

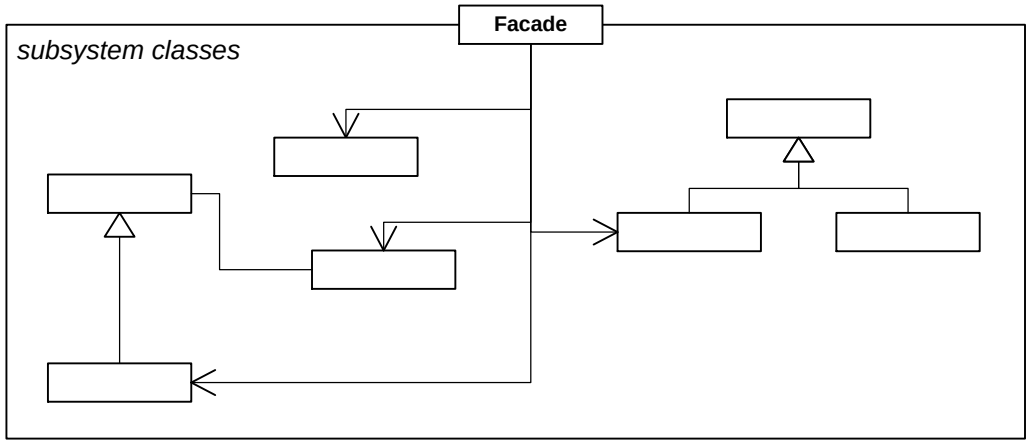
- Sie überschreiben Methoden der Oberklasse `Writer` und implementieren so ihren *Mehrwert*. Üblicherweise delegieren sie in den überschriebenen Methoden auch an die Methoden der dekorierten Instanz.

Facade-Muster

- Zweck:

Bietet eine einheitliche Schnittstelle für mehrere Schnittstellen in einem Subsystem. Facade definiert ein Interface in einem höheren Level und ermöglicht so, das Subsystem einfacher zu benutzen.

- Struktur:

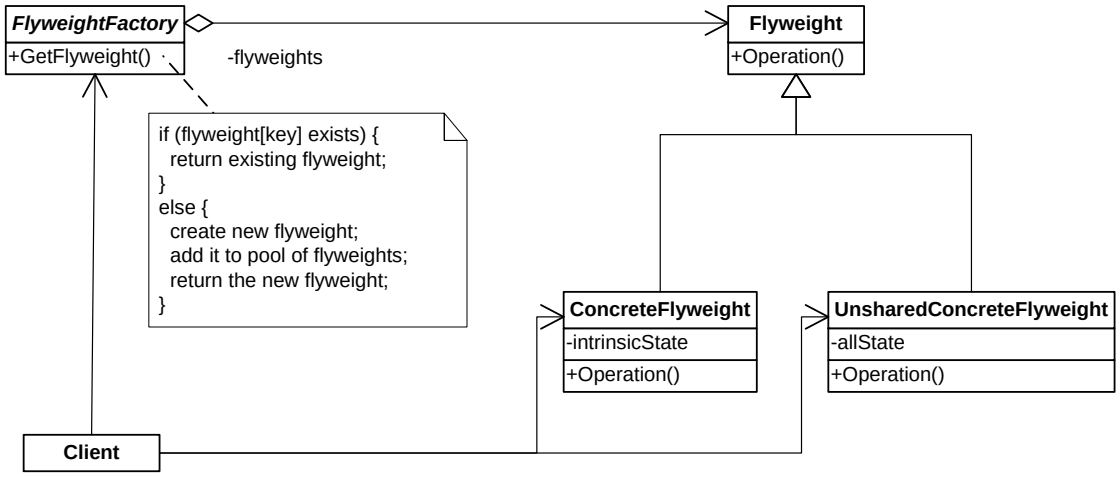


Flyweight-Muster

- Zweck:

Unterstützt Wiederverwendung, um eine große Anzahl feinkörniger Objekte effizient zu unterstützen.

- Struktur:

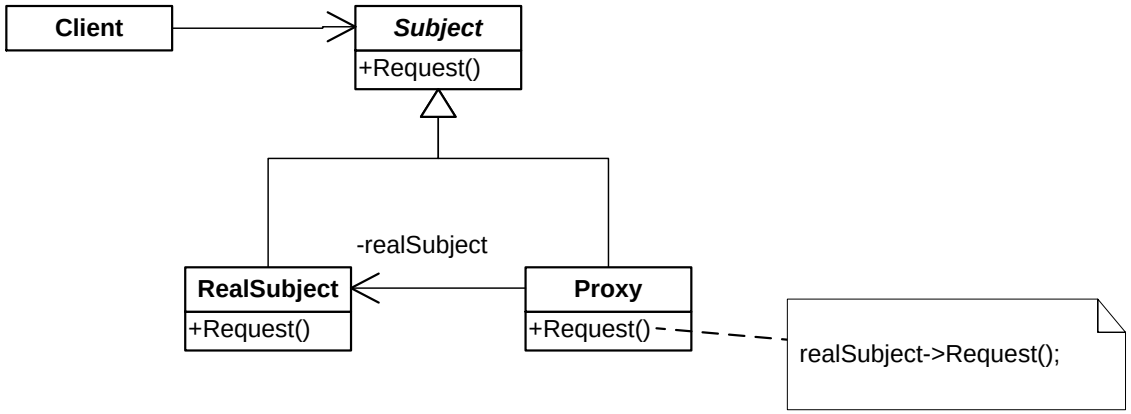


Proxy-Muster

- Zweck:

> Bietet einen Stellvertreter oder Platzhalter für ein anderes Objekt, um den Zugriff darauf zu kontrollieren.

- Struktur:



Proxy-Muster

- Anwendbarkeit:

1. **Remote-Proxy**: Bietet eine lokale Repräsentation eines Objekts in einem anderen Adressbereich.
2. **Virtual-Proxy**: Erstellt ressourcenintensive Objekte bei Bedarf. Z.B. bei Zugriff auf ein Bild mit hoher Auflösung („ImageProxy“).
3. **Protection-Proxy**: Kontrolliert Zugriff auf das Originalobjekt.
4. **Smart-Reference**: Ergänzt zusätzliche Funktionalität beim Zugriff auf ein Objekt. Es zählt z.B. die Anzahl der Referenzen und erlaubt ggf. Schreibzugriff nur für eine Referenz.

Entwurfsmuster

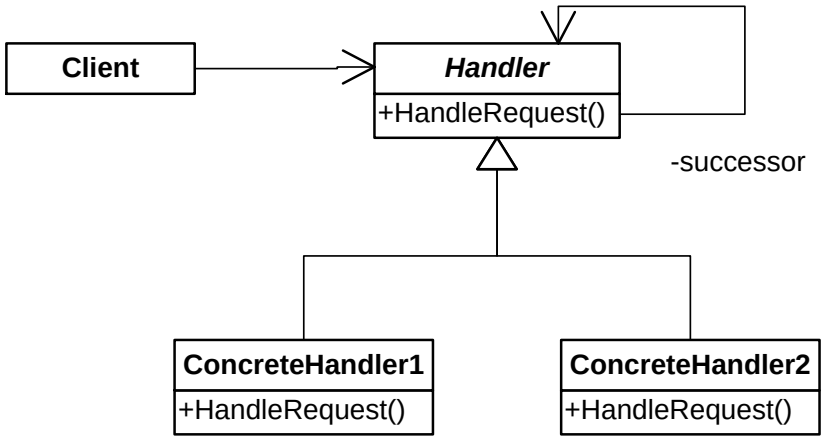
- Creational-Patterns
  - Abstract-Factory
  - Builder
  - Factory-Method
  - Prototype
  - Singleton
  - Object-Pool
- Structural-Patterns
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Facade
  - Flyweight
  - Proxy
- **Behavioral-Patterns**
  - **Chain-of-Responsibility**
  - **Command**
  - **Interpreter**
  - **Iterator**
  - **Mediator**
  - **Memento**
  - **Observer**
  - **State**
  - **Strategy**
  - **Template-Method**
  - **Visitor**

Chain-of-Responsibility-Muster

- Zweck:

Vermeidet die Kopplung vom Sender einer Anfrage mit dessen Empfänger, indem mehr als einem Objekt die Möglichkeit gegeben ist, die Anfrage zu erledigen. Verkettet die empfangenden Objekte und führt die Anfrage an der Kette entlang, bis ein Objekt sie bearbeitet.

- Struktur:

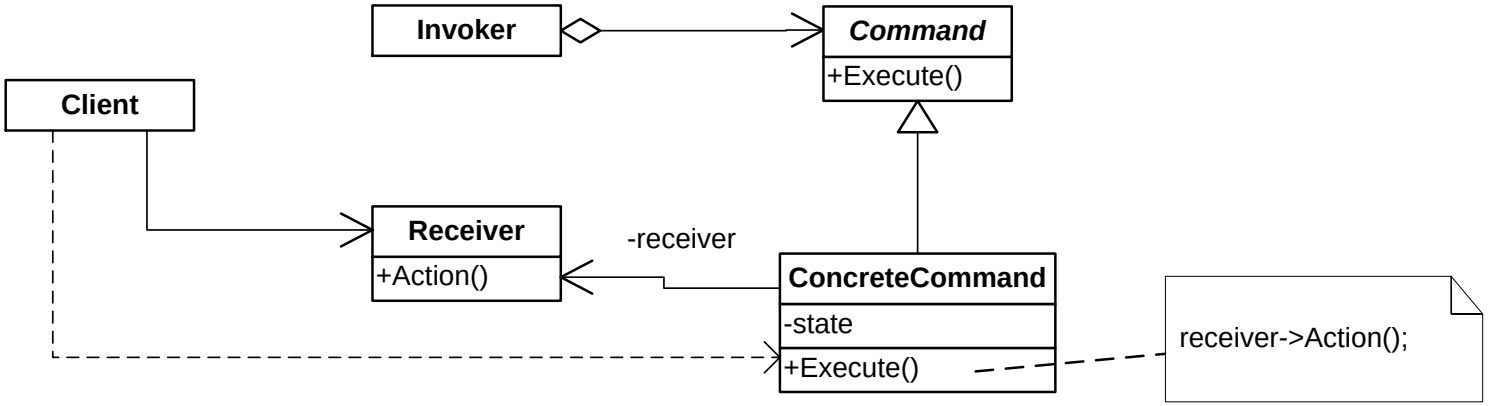


Command-Muster

- Zweck:

> Kapselt eine Anfrage als ein Objekt, wodurch ermöglicht wird, Clients mit unterschiedlichen Anfragen zu parametrisieren und Undo-Operationen zu unterstützen.

- Struktur:

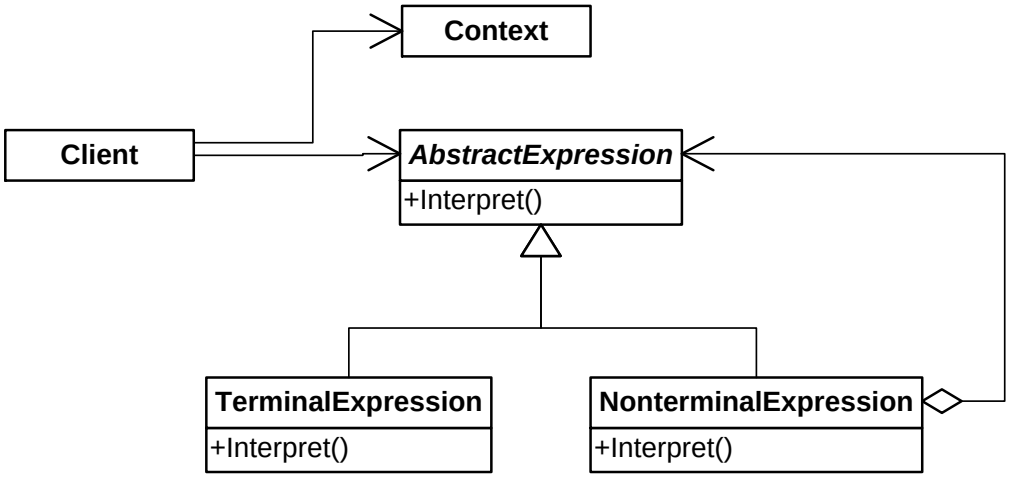


Interpreter-Muster

- Zweck:

  Definiert eine Repräsentation der Grammatik einer gegebenen Sprache, mit der Sätze dieser Sprache interpretiert werden können.

- Struktur:

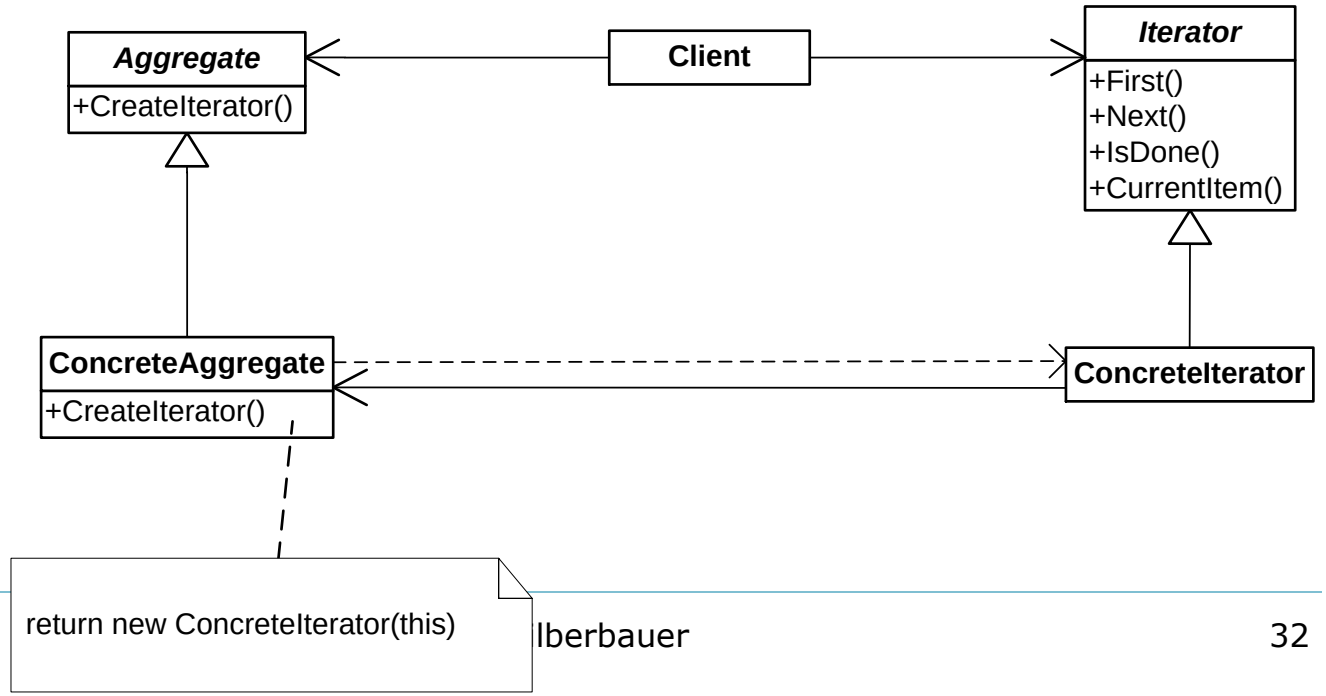


Iterator-Muster

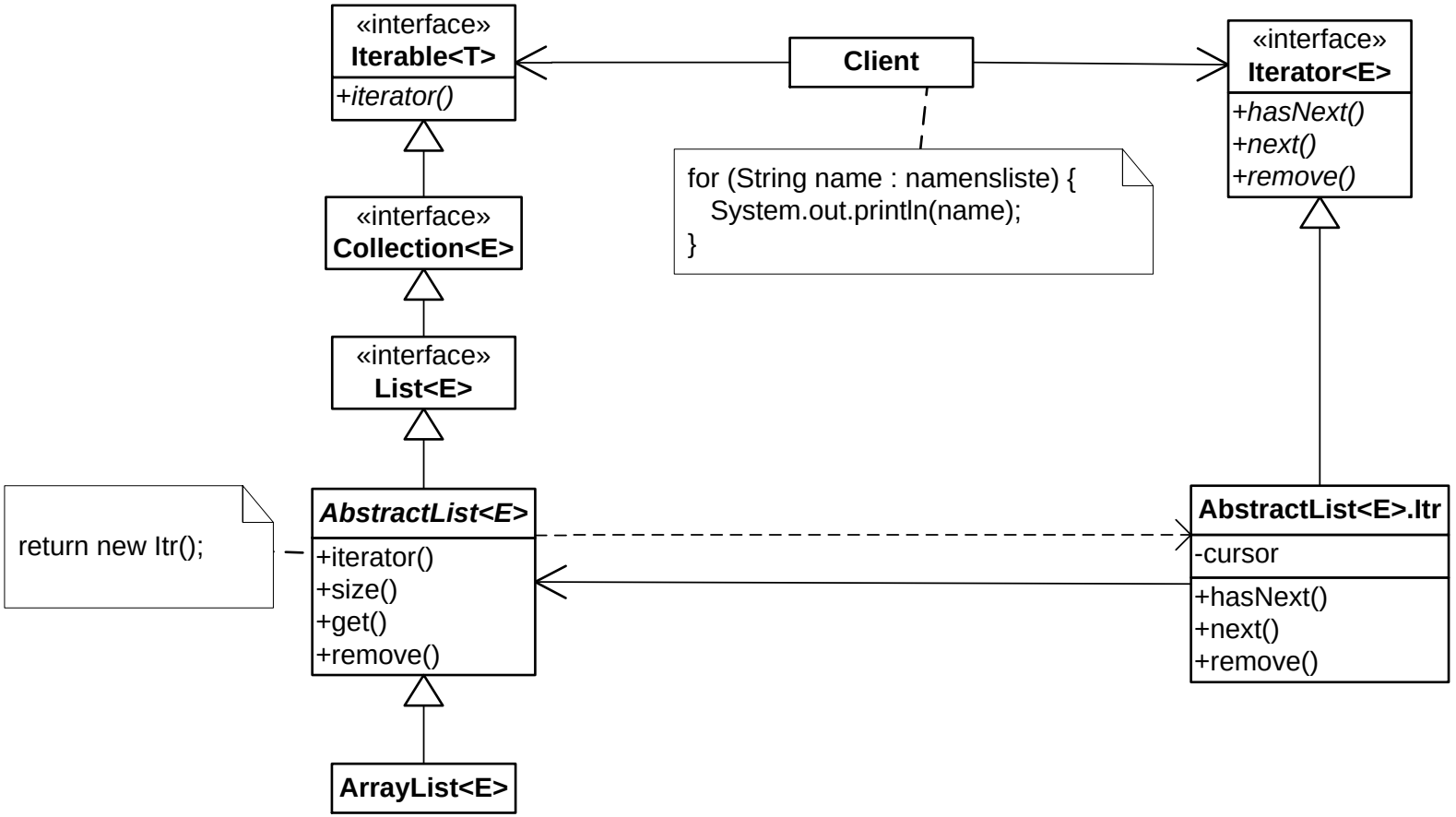
- Zweck:

  Bietet eine Möglichkeit, auf die Elemente in einem Aggregat-Objekt sequentiell zuzugreifen, ohne die zu Grunde liegende Implementierung zu offenbaren.

- Struktur:



Beispiel: Java-`ArrayList<E>`

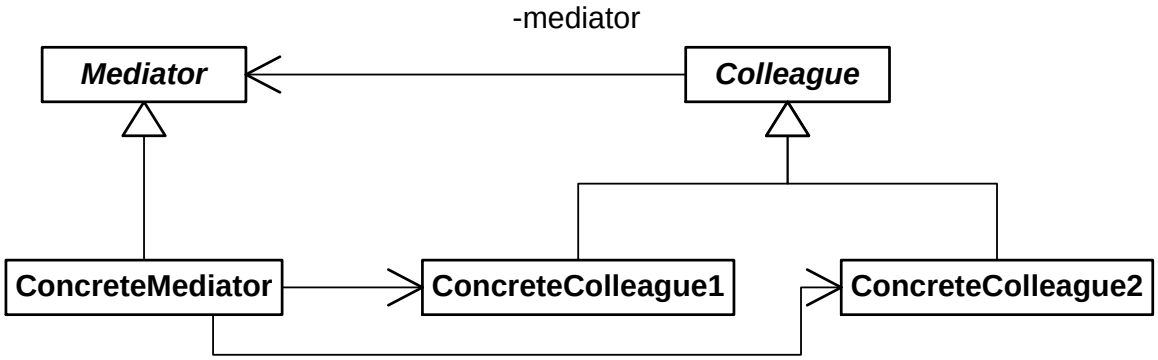


Mediator-Muster

- Zweck:

Definiert ein Objekt, das kapselt, wie eine Gruppe von Objekten untereinander interagieren. Mediator fördert eine weiche Kopplung, indem es Objekte davon abhält, sich direkt aufeinander zu beziehen und es erlaubt beliebige unabhängig Interaktionen.

- Struktur:

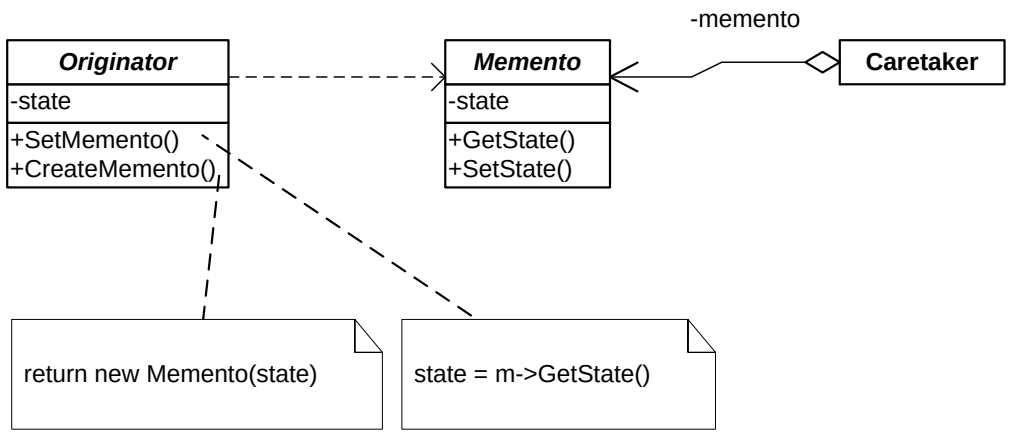


Memento-Muster

- Zweck:

Ohne die Kapselung zu verletzen, wird der interne Status eines Objekts extern gehalten, so dass das Objekt mit diesem Status später wiederhergestellt werden kann.

- Struktur:



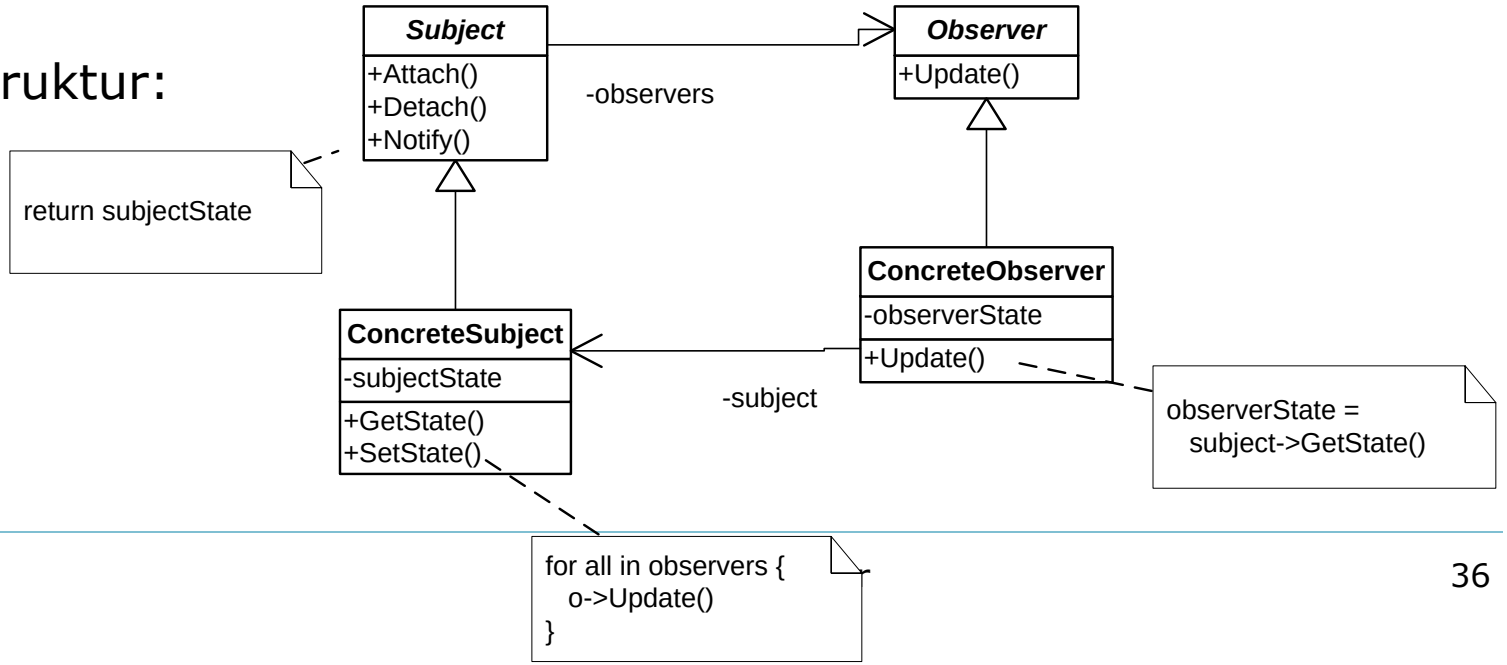
Observer-Muster

- Zweck:

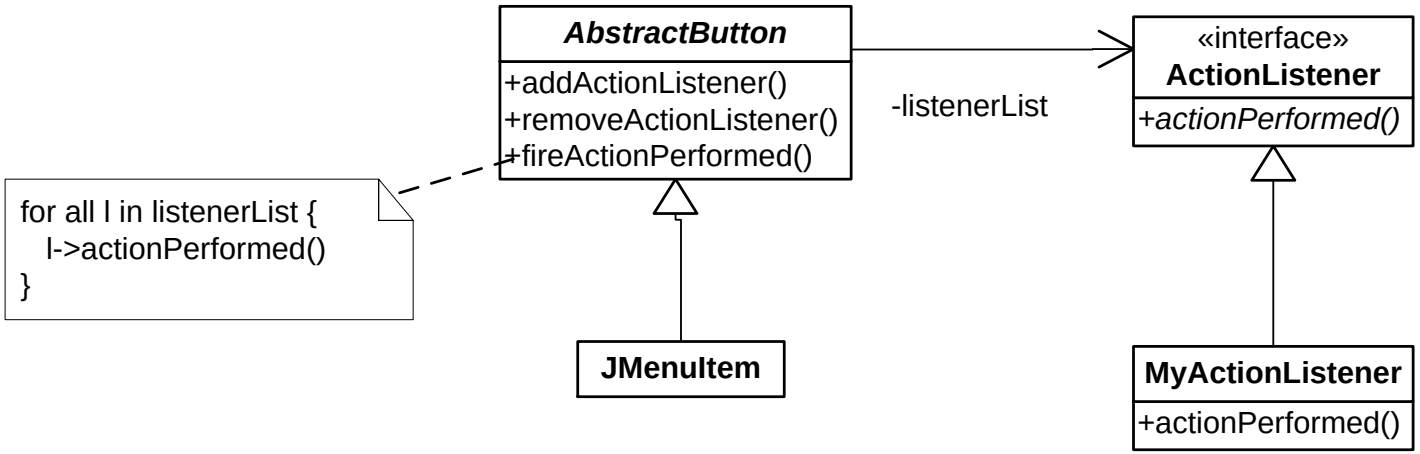
  Definiert eine Eins-zu-viele-Abhängigkeit zwischen Objekten in der Art, dass alle abhängigen Objekte benachrichtigt werden, wenn sich der Zustand des einen Objekts ändert.

- Aka: Publish-Subscriber

- Struktur:



Beispiel `JMenuItem` - `ActionListener`



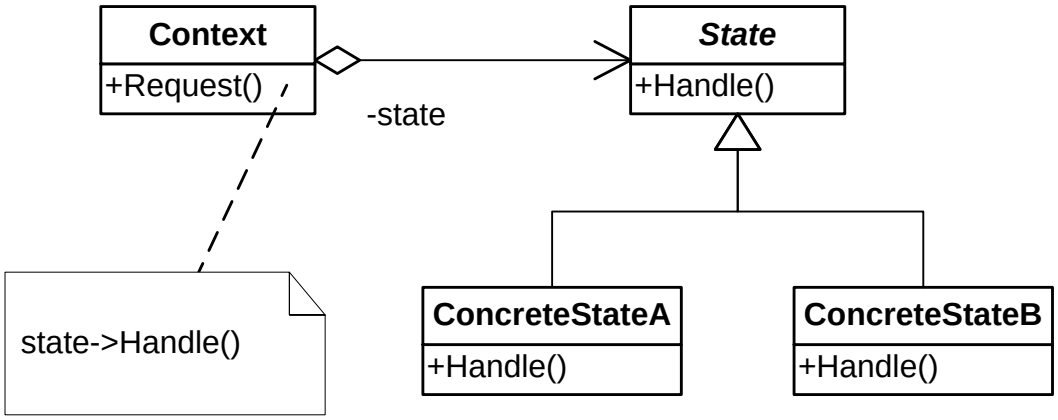
- `MyActionListener` benötigt von Vorneherein keine Referenz auf das `JMenuItem`. Er erhält diese als Argument von `actionPerformed()`. Der Parameter vom Typ `ActionEvent` erlaubt den Zugriff per `getSource()` (I.d.R. wird dieser Zugriff bei ActionListenern aber ohnehin nicht gebraucht.)
- Statt `MyActionListener` ist der *ConcreteObserver* eigentlich die anonyme innere Klasse in `KundenVerwaltung`

State-Muster

- Zweck:

  Erlaubt einem Objekt sein Verhalten anzupassen, wenn dessen interner Zustand sich ändert.

- Struktur:

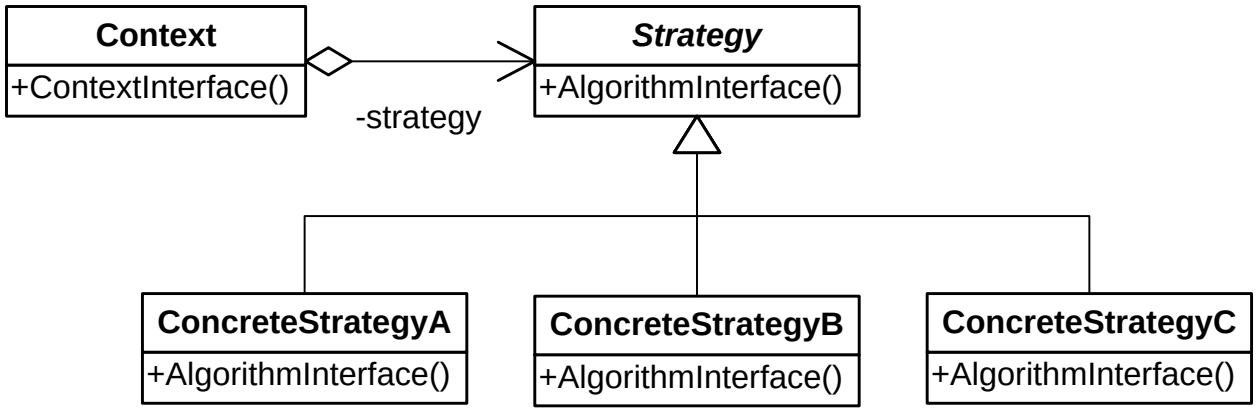


Strategy-Muster

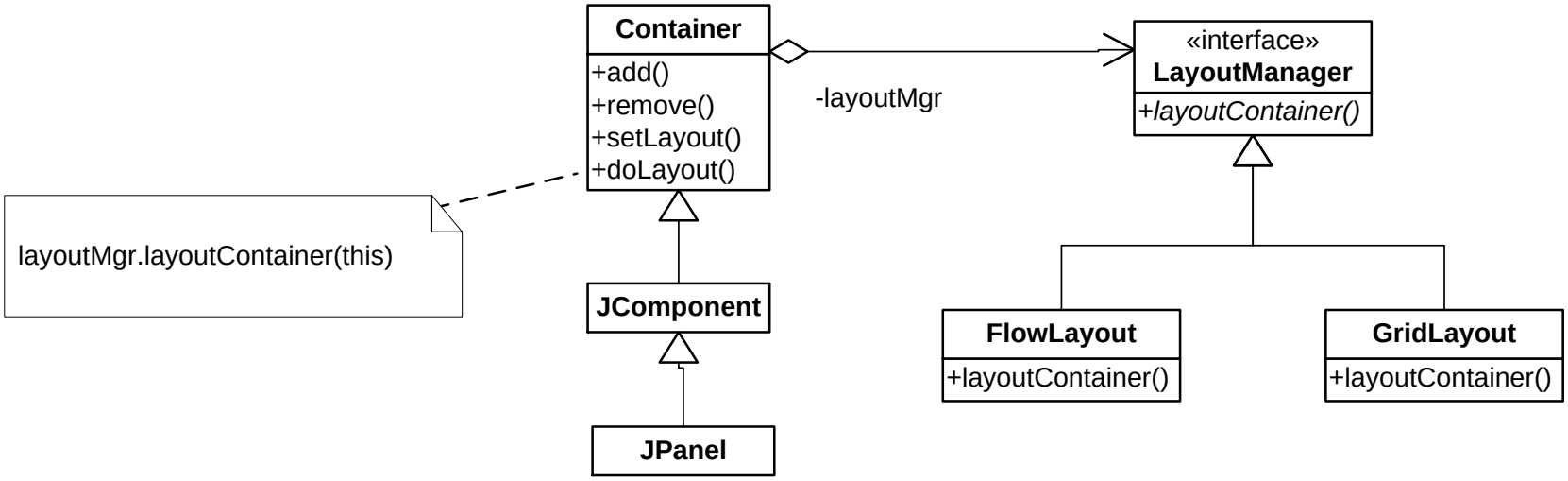
- Zweck:

> Definiert eine Familie von Algorithmen, kapselt sie einzeln und macht sie austauschbar. Strategy ermöglicht es, den Algorithmus unabhängig von den Clients die ihn einsetzen, variieren zu lassen.

- Struktur:



Beispiel `LayoutManager`



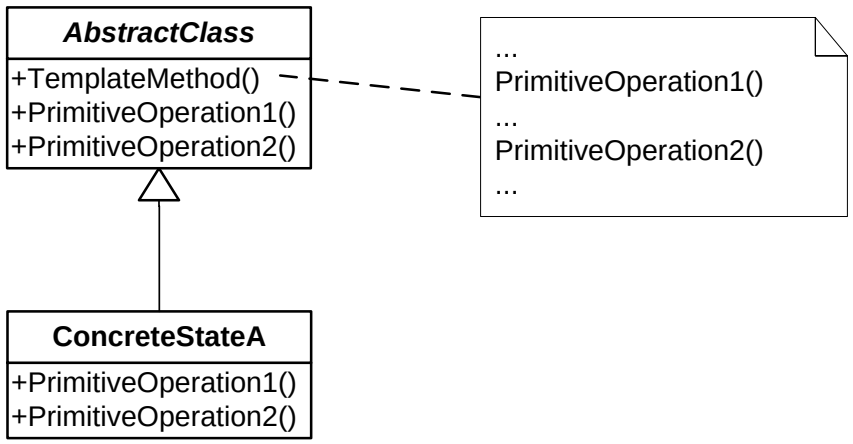
- `LayoutManager` sind *Strategien* für beliebige `Container` wie z.B. `JPanel` zur Bestimmung ihres Layouts (Position, Größe etc.) und deren Komponenten.
- `LayoutManager` bieten zum Festlegen des Layouts die Methode `layoutContainer()` an.

Template-Method-Muster

- Zweck:

> Definiert das Skelett eines Algorithmus in einer Operation, das in einigen Schritten auf Subklassen verweist. Template Method erlaubt Subklassen bestimmte Schritte eines Algorithmus neu zu definieren, ohne die Struktur des Algorithmus zu ändern.

- Struktur:

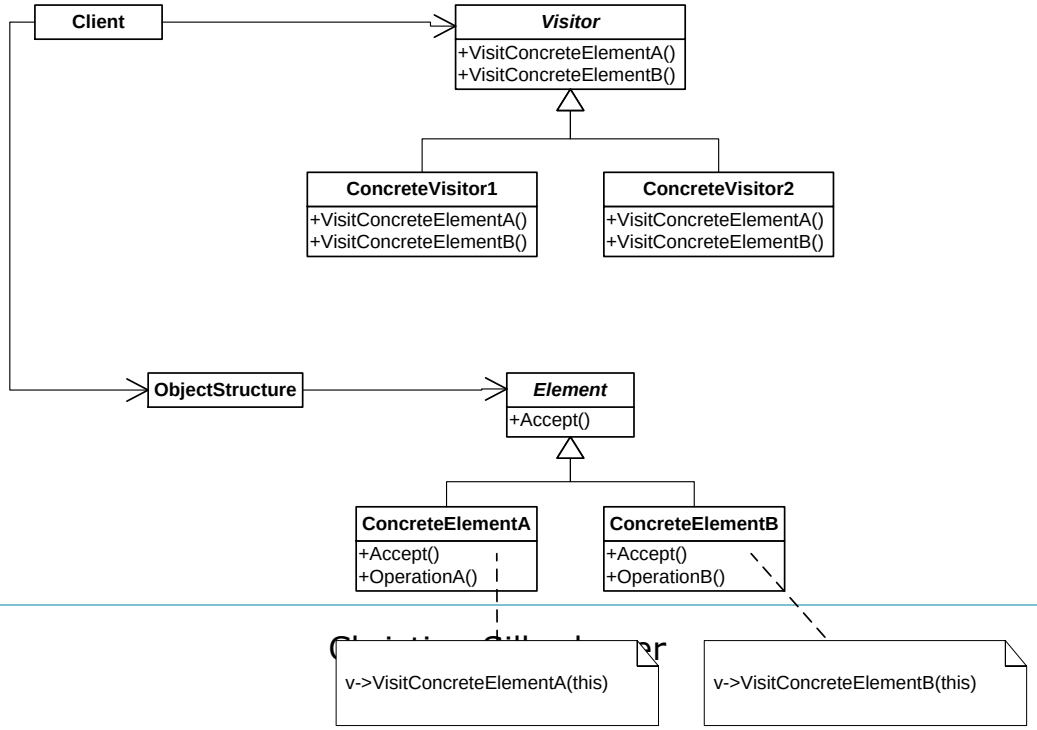


Visitor-Muster

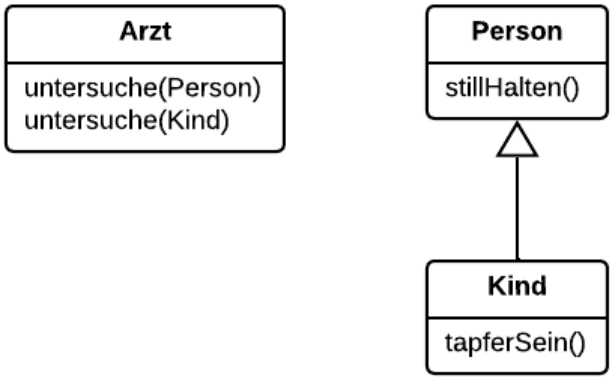
- Zweck:

> Repräsentiert eine Operation, die zur Ausführung durch Elemente eine Objektstruktur dienen. Visitor ermöglicht es, neue Operationen zu definieren, ohne diese aufrufenden Elementklassen ändern zu müssen.

- Struktur:



Beispiel: Kinderarzt



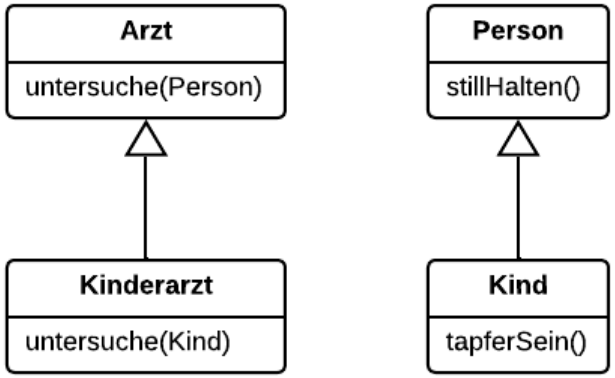

```
public class Main {
    public static void main(String[] args) {
        Arzt arzt = new Arzt();
        Person person = new Kind();
        arzt.untersuche(person);
    }
}
```

```
public class Arzt {
    public void untersuche(Person person) {
        person.stillHalten();
    }
    public void untersuche(Kind kind) {
        kind.stillHalten();
        kind.tapferSein();
    }
}
```

- `untersuche(Kind)` soll aufgerufen werden.
- `untersuche(Person)` wird aufgerufen, weil das Objekt als `Person` deklariert ist.

- Quelle: Beispiel angelehnt an https://de.wikipedia.org/wiki/Kovarianz_und_Kontravarianz, abgerufen am 2023-06-17.

Beispiel: Kinderarzt

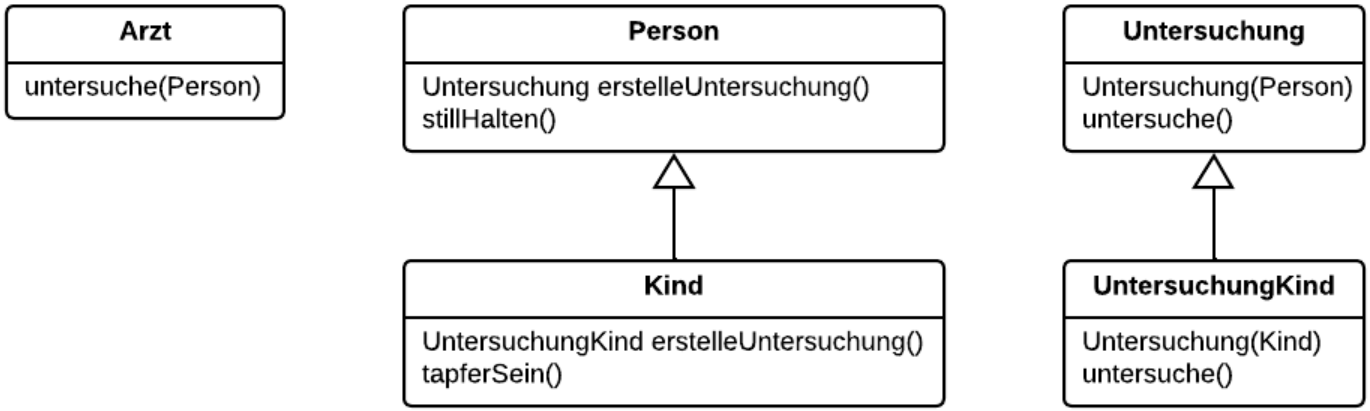


- `untersuche(Kind)` soll aufgerufen werden.
- `untersuche(Person)` wird aufgerufen, weil das Objekt als `Person` deklariert ist.

```
public class Main {
   public static void main(String[] args) {
      Arzt arzt = new Kinderarzt();
      Person person = new Kind();
      arzt.untersuche(person);
   }
}
```

```
public class Arzt {
   public void untersuche(Person person) {
      person.stillHalten();
   }
   public void untersuche(Kind kind) {
      kind.stillHalten();
      kind.tapferSein();
   }
}
```

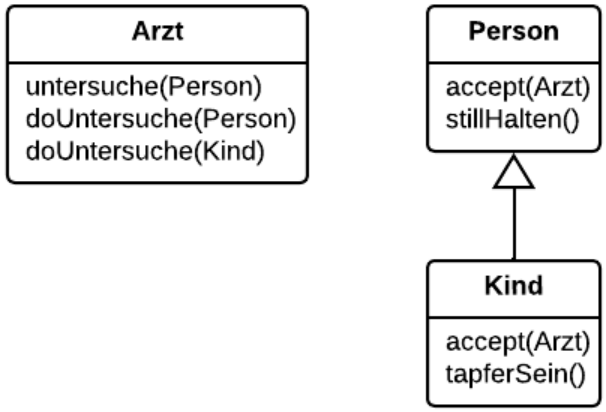
Beispiel: Kinderarzt




```
public class Arzt {
    public void untersuche(Person person) {
        Untersuchung untersuchung = person.erstelleUntersuchung();
        untersuchung.untersuche();
    }
}
```

- Funktioniert!

Beispiel: Kinderarzt




```
public class Kind extends Person {
    public void accept(Arzt arzt) {
        arzt.doUntersuche(this);
    }
    public void tapferSein() {
        System.out.println("tapfer sein");
    }
}
```

```
public class Arzt {
    public void untersuche(Person person) {
        person.accept(this);
    }
    public void doUntersuche(Person person) {
        person.stillHalten();
    }
    public void doUntersuche(Kind kind) {
        kind.stillHalten();
        kind.tapferSein();
    }
}
```

- Funktioniert auch!

Double-Dispatch

- Die Bindung der Methodenparameter erfolgt in Java statisch.
- Das heißt, gebunden wird stets an den deklarierten Typ.

- (Dynamische) Methodenbindung aufgrund des Objekttyps erfolgt nur für das Objekt selbst.
- Java unterstützt damit Einfach-Polymorphie bzw. Single-Dispatch.

- Durch das Visitor-Pattern wird Double-Dispatch unterstützt, also zusätzlich dynamische Bindung des eigentlich ersten Methodenparameters.
- Eine Verallgemeinerung dessen wäre Multiple-Dispatch oder Mehrfach-Polymorphie.
- Dies wird z.B. unterstützt in den Programmiersprachen Groovy, Julia und Raku.

Bewertung

Strukturelle Aspekte

Aspekt	Muster	Einordnung
Adaptieren	Abstract-Factory, Adapter, Bridge	• Zusammenführen verschiedenartiger, "inkompatibler" Schnittstellen • Adaption ist notwendig durch Anbindung von Legacy-Systemen • Auf ausufernde Komplexität sollte hier besonders geachtet werden. Sie sollte möglichst vermieden werden
Dependency-Injection	Alle außer: Singleton, Object-Pool, Facade, Memento	• Fundametales OOP-Muster, welches durch Vererbung und Polymorphie ermöglicht wird
"Überschreiben/ Verfeinern		• Geht einher mit Dependecy-Injection • Ist also eigentlich kein separater Aspekt
Whole/Part	…	• In OOP per Assoziation unterstützt • Ist also kein wesentlicher separater Muster-Aspekt

Strukturelle Aspekte

Aspekt	Muster	Einordnung
Strategy	Strategy, State, Memento	• Elementares Muster • Im ESC-Framework durch Expansions unterstützt
Baumstruktur/ Composite	Composite, Interpreter	• Elementares Muster • Könnte im ESC-Framework unterstützt werden, aber hätte nur geringen Mehrwert
Erweitern/ Decorator	Decorator	• Elementares Muster • Im ESC-Framework durch Extensions unterstützt

Strukturelle Aspekte

Aspekt	Muster	Einordnung
Hardwarebezug	Singleton, Object-Pool, Flyweight, Proxy, Memento	Teilaspekte: • Ressourcen sparen (Laufzeit/Speicher) • darunter auch "Spätes Laden" (Proxy, Singleton und Object-Pool) • Ortstransparenz (Proxy) • Synchronisierung bei Nebenläufigkeiten (Proxy, Object-Pool) SimPL ingoriert Hardwarebezug, im SimPL-Interpreter werden Flyweight, Proxy und Memento aber angewendet.
Ereignis-verarbeitung	Chain-of-Responsibility, Command	• Wird typischerweise verwendet in OO-Frameworks zur Kapselung der Kommunikation (z.B. in UI-Frameworks) • Im ESC-Framework verwendet durch die Ereignis-Prozessoren

Strukturelle Aspekte

Aspekt	Muster	Einordnung
Iterator	Iterator	• Von Programmiersprachen unterstützt, z.B. Erweiterte For-Schleife in Java
Benachrichtigungen/Observer	Observer	• Elementares Muster • Im ESC-Framework durch autoDelegate unterstützt
Double-Dispatch	Visitor	• Elementares Muster • Muster notwendig in OO-Sprachen mit Single-Polymorphism • Das ESC-Framework deckt das ab, indem es Mehrfach-Polymorphie unterstützt.

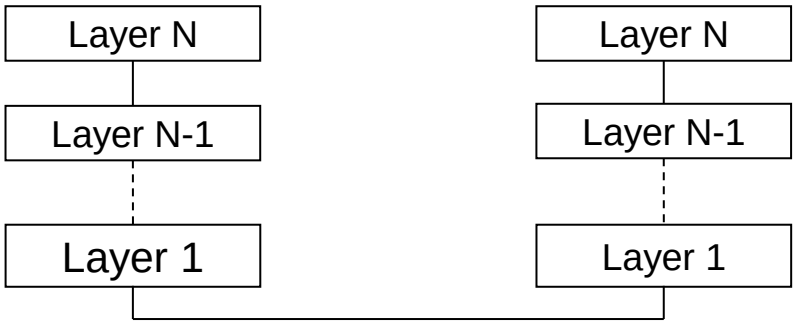
Beschreibung von Architekturmustern

- Struktur eines Systems
  - Layers
  - Pipes and Filters
- Adaptive Systeme
  - Plug-in
- Verteilte Systeme
  - Broker
  - Service-Oriented Architecture
- Interaktive Systeme
  - Model-View-Controller

Layers

- (Buschmann, 1996):

> The Layers architectural pattern helps to structure applications that can be decomposed into groups of subtasks in which each group of subtasks is at a particular level of abstraction.



- (Goll, 2014):

> Das Architekturmuster Layers modelliert die Struktur eines Systems in Schichten mit Client/Server-Beziehungen zwischen den Schichten.

Pipes and Filters

- (Buschmann, 1996):

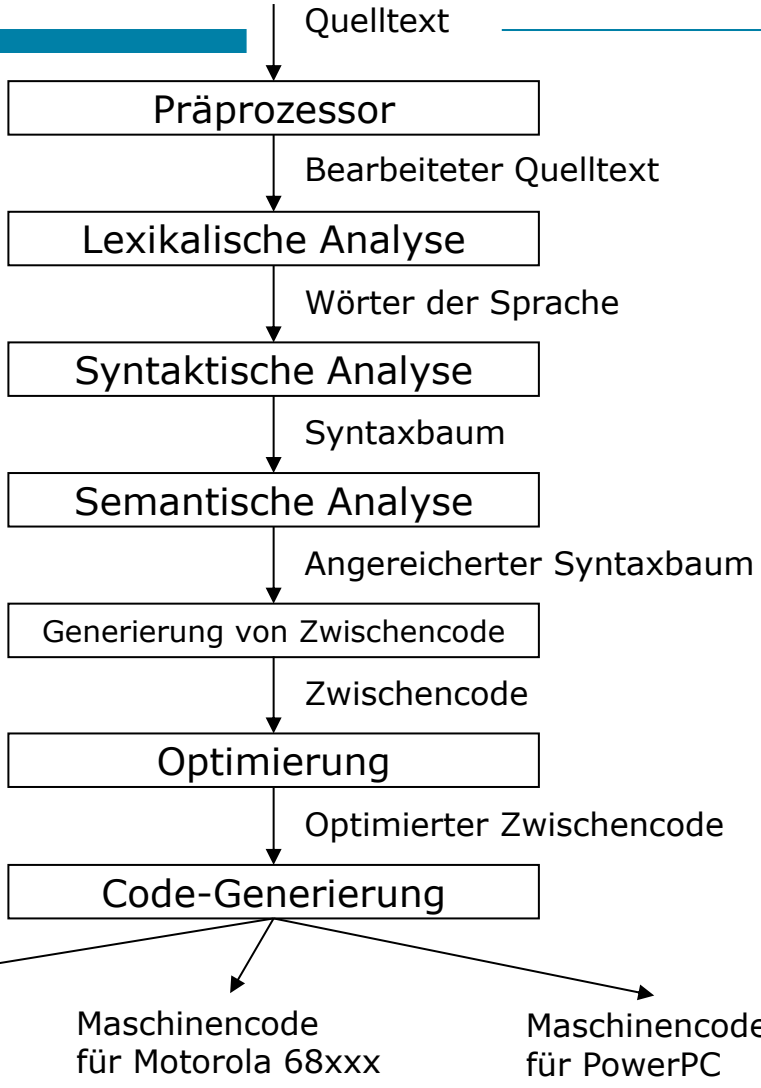
> Provides a structure for system that process a stream of data. Each processing step is encapsulated in a filter component. Data is passed through pipes between adjacent filters. Recombining filters allow you to build families of related systems.

- (Goll, 2014):

> Die Aufgabe des gesamten Systems wird in einzelne Verarbeitungsstufen zerlegt. Jeder Verarbeitungsschritt wird in Form eines **Filters** implementiert. Jeweils zwei Filter werden durch eine Pipe verbunden. **Filter** lesen die Daten und bearbeiten sie sequenziell. Ein Filter wandelt eine Dateneingabe in eine Datenausgabe um.

Pipes and Filters

- Beispiel:

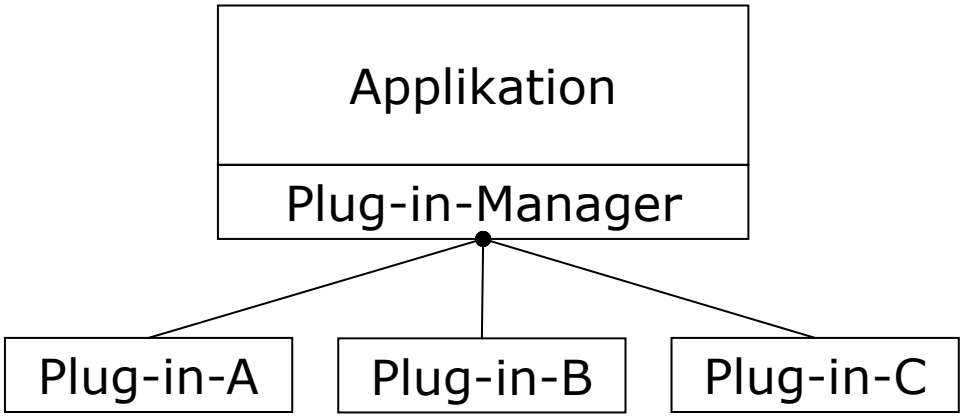


Plug-in

- (Goll, 2014):

> Strukturiert eine spezielle Anwendung so, dass diese über Erweiterungspunkte in Form von Schnittstellen verfügt, an denen dann die Plug-ins, die diese Schnittstellen implementieren, eingehängt werden können. Eine Anwendung ist aber bereits ohne diese zusätzlichen Erweiterungen lauffähig. Die Architektur der Anwendungssoftware muss Schnittstellen definieren, an deren Stelle **zur Laufzeit Komponenten, die diese Schnittstelle implementieren**, treten können.

- Struktur:



Broker

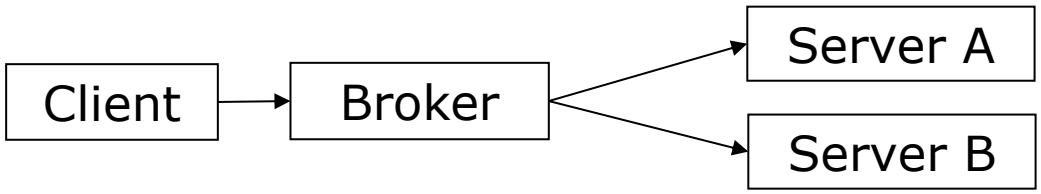
- (Buschmann, 1996):

> Can be used to structure distributed software systems with decoupled components that interact by remote service invocations. A broker component is responsible for coordinating communication, such as forwarding requests, as well as for transmitting results and exceptions.

- (Goll, 2014):

> Entkoppelt in verteilten Softwarearchitekturen Client- und Server-Kommunikation, indem zwischen diesen Komponenten ein Broker als Zwischenschicht eingeschoben wird und Client- und Server-Komponente untereinander nur über diesen Broker kommunizieren.

- Struktur:



Service-Oriented Architecture (SOA)

- (Goll, 2014):

> Kapselt die Verarbeitungsfunktionen einer Software, die Geschäftsprozessen oder Teilen von Geschäftsprozessen entsprechen, als Dienste in Komponenten bzw. Systemteilen.

- (OASIS, 2006):

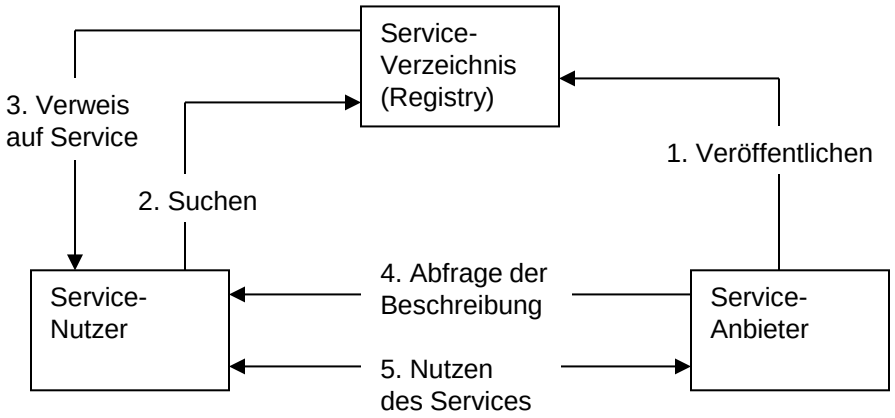
> Service Oriented Architecture (SOA) is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. It provides a uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectations.

Service-Oriented Architecture

- Schlüsselkonzepte:
  - **Sichtbarkeit** *ermöglicht es jenen mit Bedürfnissen und jenen mit Fähigkeiten, sich gegenseitig wahrzunehmen.* Üblicherweise wird das durch Beschreibungen (*service descriptions*) von Funktionalitäten, technischen Anforderungen usw. realisiert.
  - **Interaktion** *ist die Aktivität, eine Fähigkeit in Anspruch zu nehmen*. Typischerweise wird dies durch den Austausch von Nachrichten umgesetzt. Die Interaktionen finden in einem sog. **Ausführungskontext** (*execution context*) statt. Dieser stellt die Menge von technischen und geschäftlichen Elementen dar, die einen Pfad zwischen jenen mit Bedürfnissen und jenen mit Fähigkeiten bilden.
  - *Eine* **Wirkung** (*real world effect) ist das Ergebnis einer Interaktion*. Es kann sich dabei um eine Rückgabe von Informationen und/oder die Änderung des Zustands einer Entität handeln.

Service-Oriented Architecture

- Struktur:



- Der Vorteil von SOA ist es, die Handhabung großer, verteilter, veränderlicher betrieblicher Anwendungen zu erleichtern.
- SOA ist geprägt von Skalierbarkeit, Transparenz und loser Kopplung.

Service-Oriented Architecture

- Service nach (OASIS, 2006):

  > The means by which the needs of a consumer are brought together with the capabilities of a provider.

- Ein Service verbindet folgende Grundgedanken:
  - Die Fähigkeit gegenseitig Arbeit zu erbringen (Implementierung)
  - Die Spezifikation der angebotenen Arbeit (Schnittstelle)
  - Das Angebot, Arbeit zu erbringen (Registry)

- Demnach wird unterschieden, dass eine Fähigkeit vorhanden ist und dass eine solche Fähigkeit erbracht wird.

Service-Oriented Architecture

- SOA erweitert Komponenten-orientierte Architekturen:
  - (Szyperski, 1996):

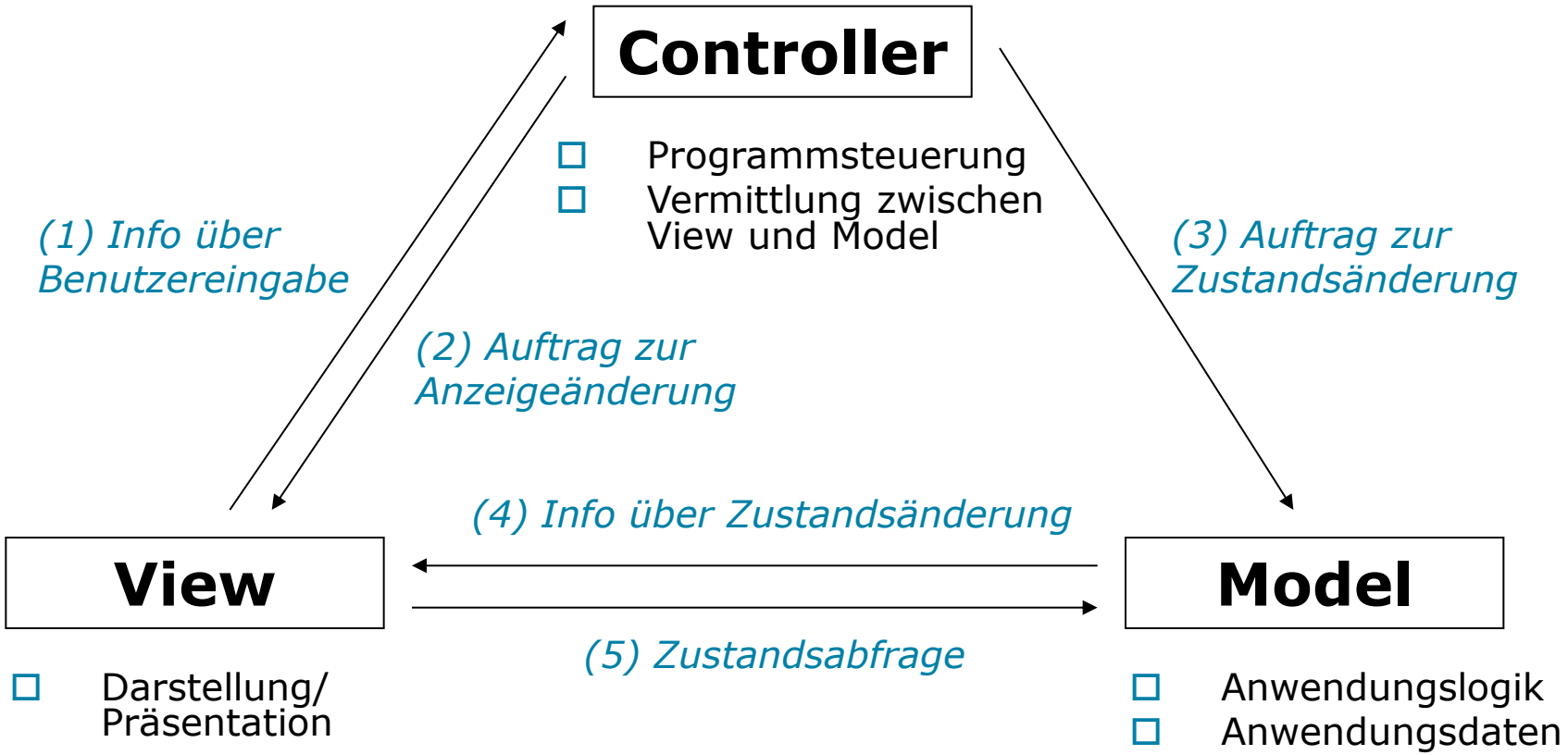
> A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

- SOA erweitert dies im Wesentlichen um das Service-Verzeichnis.

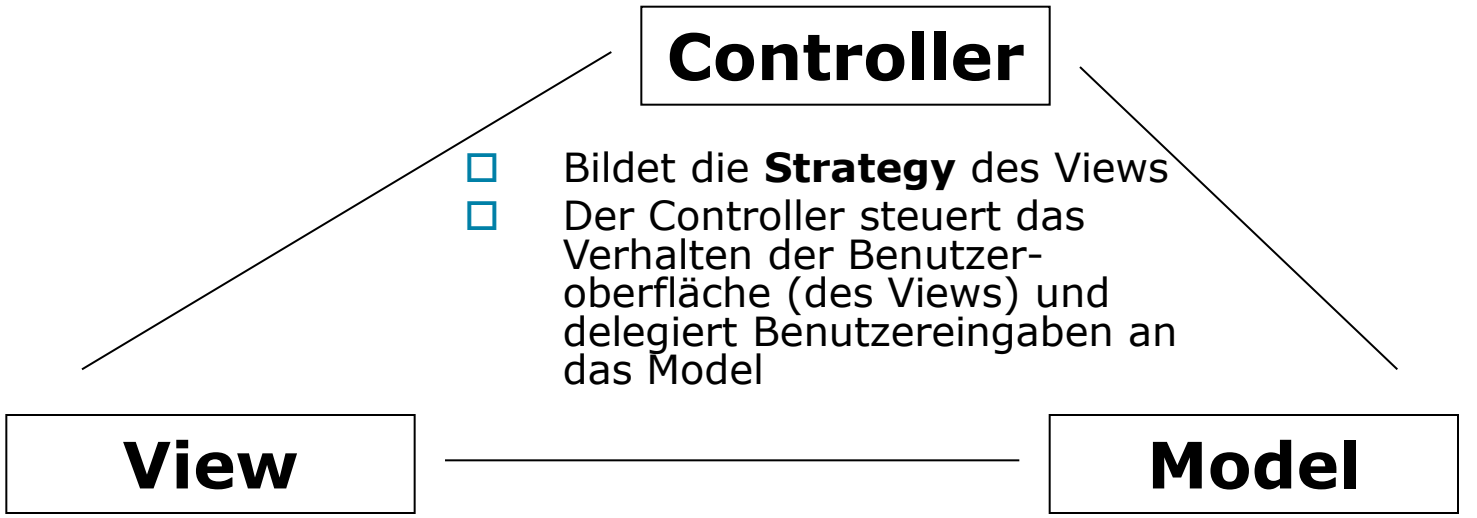
Model-View-Controller (MVC)

- Trennung von Zuständigkeiten speziell bei Anwendungsdialogen
- MVC forciert damit **Separation-of-Concerns** (SoC; siehe insb. Edsgar W. Dijkstra, 1976, A Discipline of Programming).
- Erstmals wurde MVC 1979 für die Benutzeroberflächen in Smalltalk beschrieben.
- Trennung erfolgt in **M**odel (Anwendungslogik), **V**iew (Präsentation) und **C**ontroller (Programmsteuerung)
- Die Komponenten sind untereinander lose gekoppelt (im Kern mittels Dependency-Injection).

Kommunikation bei MVC



MVC als Komposition von Entwurfsmustern

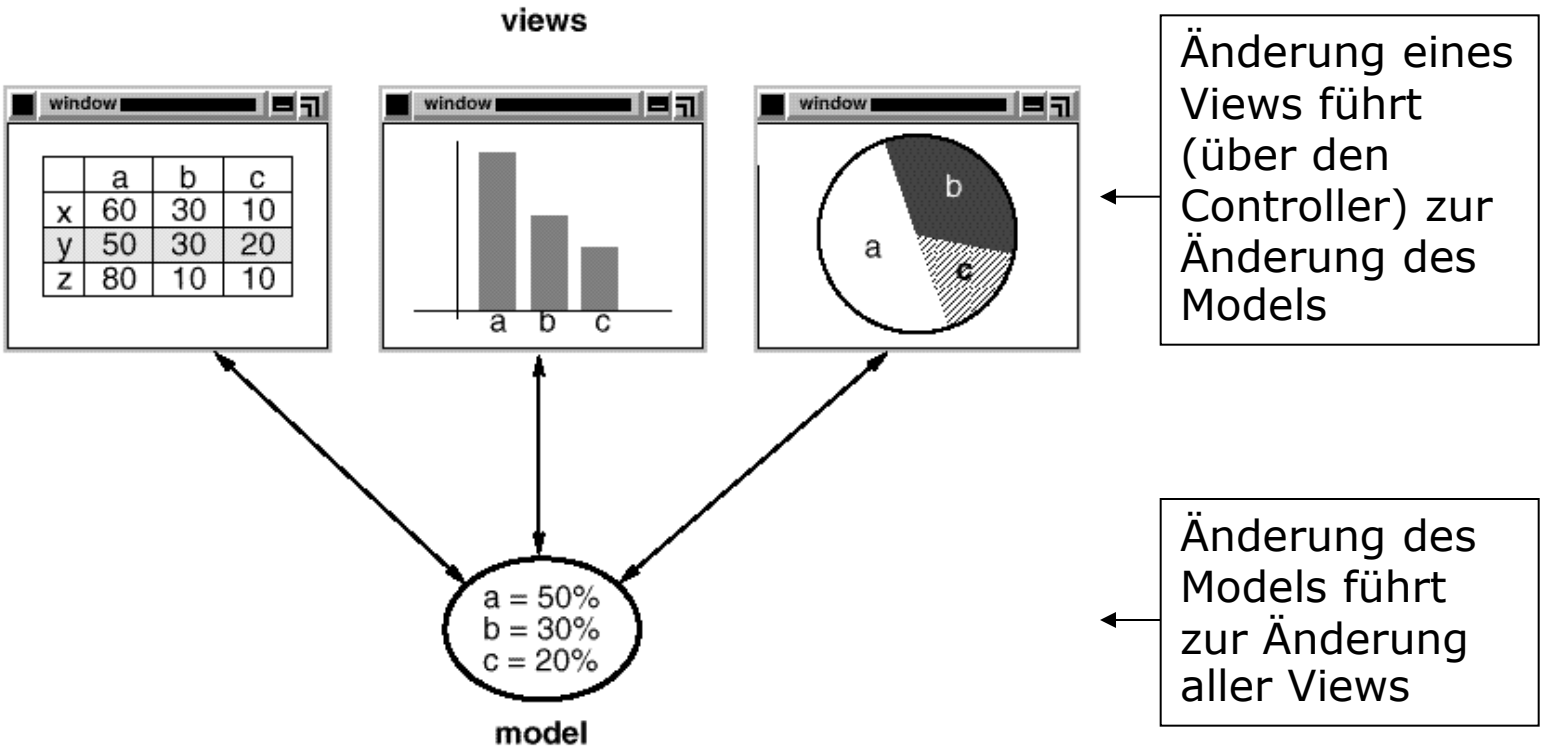


- Implementiert das **Composite**-Muster
- GUI-Komponenten bilden eine Baumstruktur
- Z.B. die Aktualisierung der Anzeige löst der Controller über die oberste relevante Komponente aus, der Rest ist Aufgabe des Composite-Patterns

- Bietet einem oder sogar mehreren Views an, sich als **Observer** für Zustandsänderungen des Models zu registrieren

Ein Model, viele Views

- Quelle: (Gof, 1994), S. 5:



Strukturelle Aspekte

Aspekt	Muster	Einordnung
Middleware	Broker, SOA	• Unterbau für die Applikationsentwicklung • Abstraktion der Hardware • Bereitgestellt durch den SimPL-Container • Unterbau des ESC-Component-Frameworks
Adaptieren	Plug-in	• Einbindung spezifischer Funktionalität • Bereitgestellt durch den SimPL-Container • Unterbau des ESC-Component-Frameworks
Interaktion	MVC	• Struktur für komplexe Interaktionen
Sequenzieren	Layers, Pipes and Filters	• Abstrakte Container zur Abarbeitung in Sequenzen

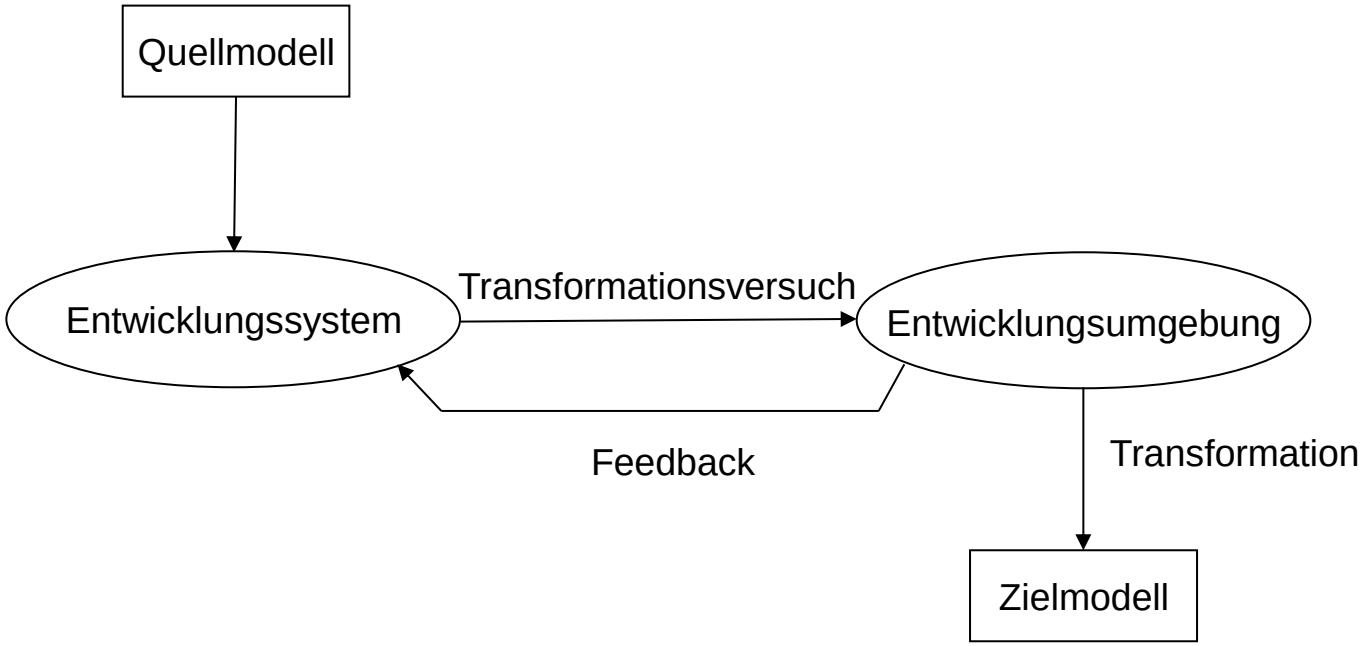
Eine Musterkritik

- Muster gibt es in der Softwareentwicklung in vielerlei Art.
- Muster sind bewährte Lösungen von Problemen in einem bestimmten Kontext.

- Praktisch wählt man aus einem Katalog von (teilweise konkurrierenden) Mustern aus und wendet diese an.
- Das (gemeinsame) Wissen über Muster erleichtert die Entwicklung.

- Allerdings: eine zielführendere, strukturiertere damit verbundene Lösung kann zu weiterer Verbesserung führen.
- Beispiele:
  - Metaprogrammierung in SimPL
  - Extensions und Expansions im ESC-Framework

Wdh.: Komplexität und Modellierung



Paradigmen

- Paradigmen sind Sichtweisen auf die Programmierung.
- Programmiersprachen unterstützten Paradigmen, indem sie syntaktische Elemente zu deren Umsetzung bereitstellen.

- Das bedeutet weder:
  - Dass ein Programm einem Paradigma folgt, nur weil die Programmiersprache dies unterstützt.
  - Und dass ein Programm nicht einem Paradigma folgt, nur weil eine Programmiersprache dieses nicht unterstützt.

- Gegenüber Architekturmuster können sie abgegrenzt werden durch einen universelleren Anwendungsbereich, womit sich eben ein Sprachunterstützung lohnt.

Paradigmen

- Im Folgenden werden vorgestellt:
  - Objektorientierte Programmierung
  - Aspektorientierte Programmierung
  - Subjektorientierte Programmierung
  - Ereignisgetriebene Programmierung
  - Zustandsorientierte Programmierung
  - Agentenorientierte Programmierung
  - Funktionale Programmierung
  - Logische Programmierung
  - Modellgetriebene Programmierung

Objektorientierte Programmierung

- Typbildung in Form von Klassen.
- Sie umfassen Methoden und Attribute.
- Instanzen von Klassen werden als Objekte bezeichnet.

- OOP ist heterogen definiert.
- Es umfasst die Basiskonzepte Datenkapselung, Vererbung und Polymorphie.
- Datenkapselung und Veerbung sind statisch, Polymorphie ist dynamisch, führt also zu Laufzeitkosten.

- Da Attribute Variablen eines Klassentyps sind, entsteht somit eine sog. Aggregationshierarchie.

Datenkapselung

- Innere Zustände werden über Methoden adressiert.
- Per Konstruktor wird ein Objekt mit gültigem Initialzustand erzeugt.
- Methoden überführen innere gültige Zustände in gültige Folgezustände.
- OO-Sprachen unterstützen die Konfiguration von Sichtbarkeit (`public`, `protected`, `private`), um Zugriffsmöglichkeiten einzuschränken.
- Datenkapselung reduziert die Notwendigkeit von Gültigkeitsprüfungen (damit Redundanz).

Vererbung

- Wiederverwendung der Elemente von *Oberklassen* bzw. *Basisklassen*.
- Sie werden *geerbt*.
- Somit werden Typhierarchien gebildet.
- Vererbung reduziert Redundanz.

Polymorphie

- Ermöglicht dynamische Methodenbindung unter Einhaltung der Typsicherheit.
- Umgesetzt wird diese mittels Virtueller Methodentabellen (VMT), aka „Virtual Function Table“.
- Das Objekt referenziert dazu die VMT vom Typ des Objekts. Diese bestimmt, welche Methoden bei einem Aufruf zum tragen kommen.
- Voraussetzung für Polymorphie ist eine Typhierarchie.
- Methoden eines konkreteren Typs (in der Hierarchie weiter unten) *überschreiben* Methoden eines allgemeineren Typs.
- Die Methode am nächsten zu eigentlichen Objekttyp kommt also zum tragen.

Aspektorientierte Programmierung

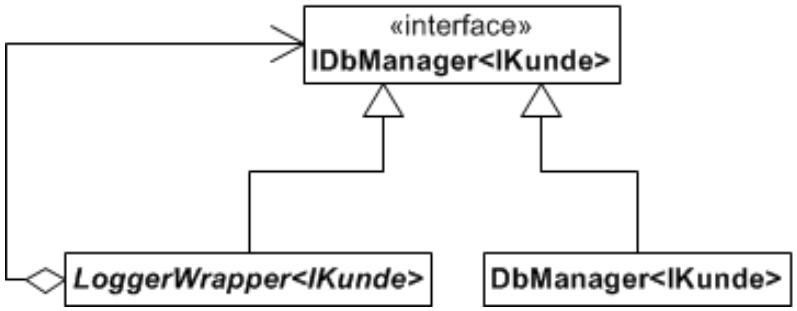
- OOP erlaubt Typ- und Aggregationshierarchien.
- AOP ergänzt typischerweise die OOP (z.B. AspectJ).
- Durch Weaving erlaubt sie Code für Methoden zu ergänzen.
- Somit ist dem Bedarf an Mehrdimensionalität von Softwaresystemen Rechnung getragen.
- Sog. Querschnittsbelange wie Concurrency, Logging oder Security können so „eingewebt" werden.
- Alternativen sind das einhängen von Querschnittsbelangen durch Frameworks oder Codegeneratoren.
- Aspektorientierte Sprachen leiden oftmals unter mangelnder IDE-Unterstützung. Die Mehrdimensionalität ist so nur schwer erkennbar.
- Zudem fehlt es an Typsicherheit.

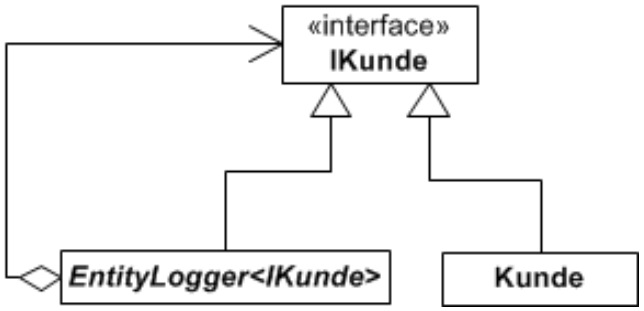
Beispiele

- Methodenaufrufe sollen bei Entity-Klassen zum Loggen führen.
- Das Persistenz-Management soll für Entity-Klassen generisch umgesetzt werden.

- Nutzt man Java für AOP, ist hierbei das Decorator-Pattern hilfreich.
- Dies wird ergänzt um den Dynamic-Proxy, der eine 1-n-Betrachtung zwischen eigentlicher Anwendung und Querschnittsbelang erlaubt.

Architektur

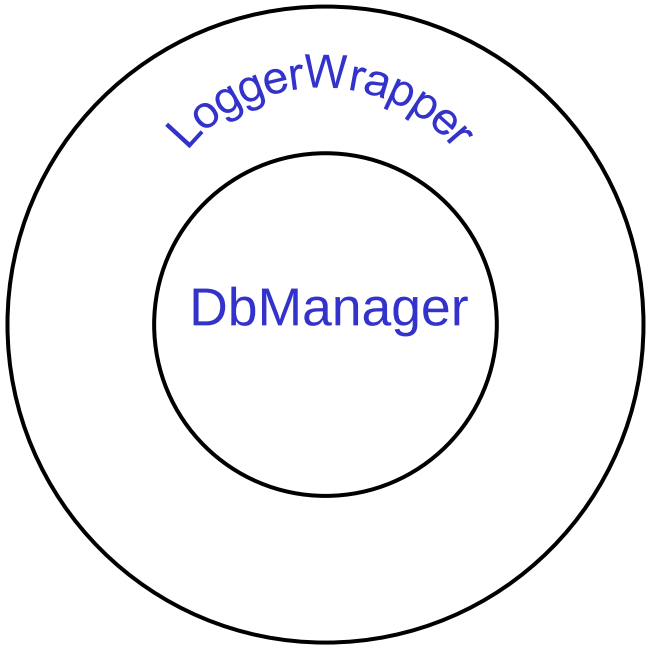


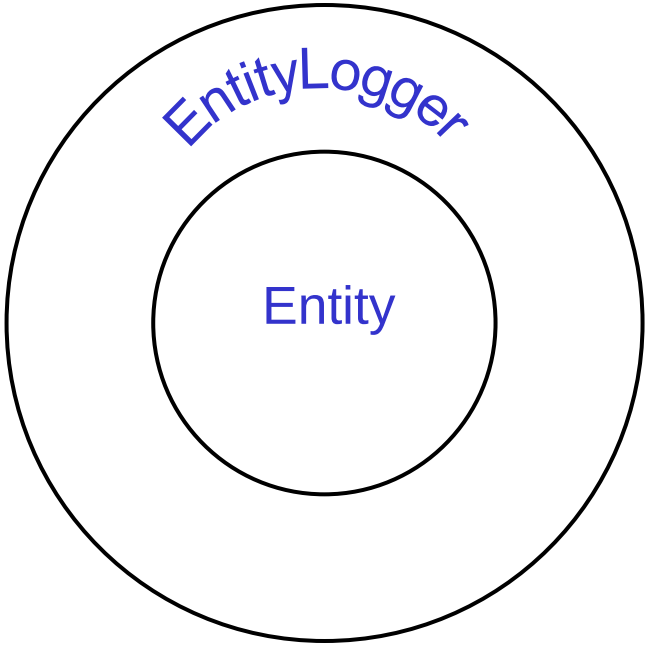


- `DbManager` wird durch `LoggerWrapper` dekoriert
- `LoggerWrapper` ist an `IDbManager` gebunden

- `Kunde` wird durch `EntityLogger` dekoriert
- `EntityLogger` ist eigentlich unabhängig von `IKunde`

Architektur

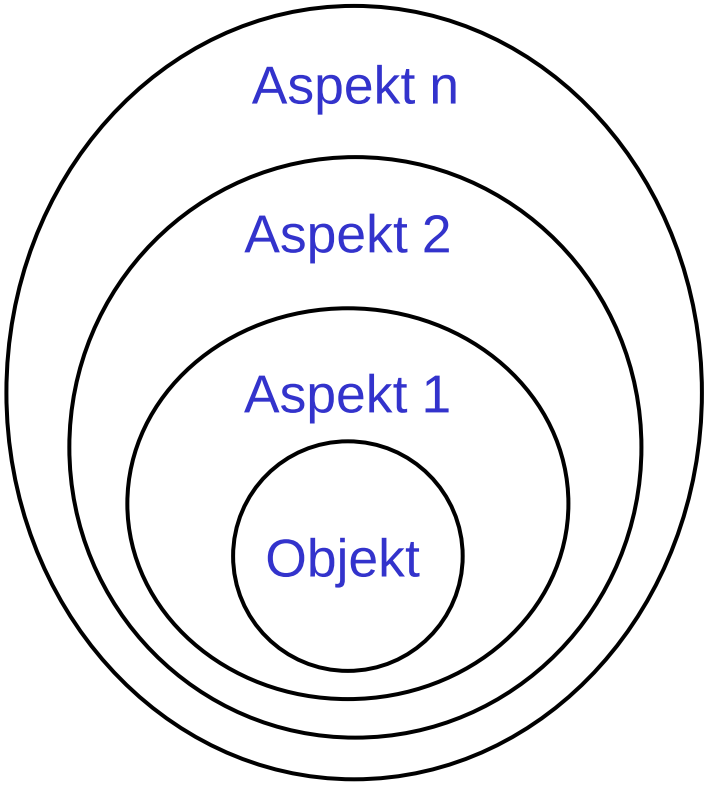




- Die Kommunikation zum `DbManager` verläuft über den `LoggerWrapper`
- Die Kommunikation zur `Entity` verläuft über den `EntityLogger`

Aspektorientierte Programmierung

- Wir haben Objekte um den Aspekt *Logging* erweitert.
- Im Allgemeinen könnten wir Objekte um mehrere *Aspekte* erweitern wollen.
- Wir wollen dabei die Kommunikation mit einem Objekt überwachen und ggf. auch manipulieren (z.b. Autorisierungs-Aspekt).



Subjektorientierte Programmierung

- Subjektorientierte Programmierung erweitert Aspektorientierte Programmierung, indem aufgerufene Funktionalität vom Aufrufer (dem Subjekt) abhängig ist.

Ereignisgetriebene Programmierung

- Ereignisgetriebene Programmierung geht davon aus, dass das Kommunikationsfluss primär durch Ereignisse bestimmt ist.
- Ereignisquellen erzeugen dabei initiale Ereignisse.
- Folgeereignisse reagieren auf Ereignisse.
- Eine 1-zu-n-Kommunikation ist dabei elementar.
- Ereignisgetriebene Programmierung geht von Zustandslosigkeit aus.
- Ereignisse können zeitabhängig sein.

Ereignisgetriebene Programmierung

- Ereignisgetriebene Programmierung kann auf Abfragesprachen wie EPL (Event Processing Lanuagage) oder CQL (Continuous Query Language) basieren.
- Somit ist Ereignisverarbeitung auf höherem Abstraktionsniveau möglich.

- Beispiel EPL:

```
INSERT INTO ta_per_hour
  SELECT symbol, COUNT(*) AS cnt
    FROM asset_ta
    RETAIN BATCH OF 1 HOUR
    GROUP BY symbol
```

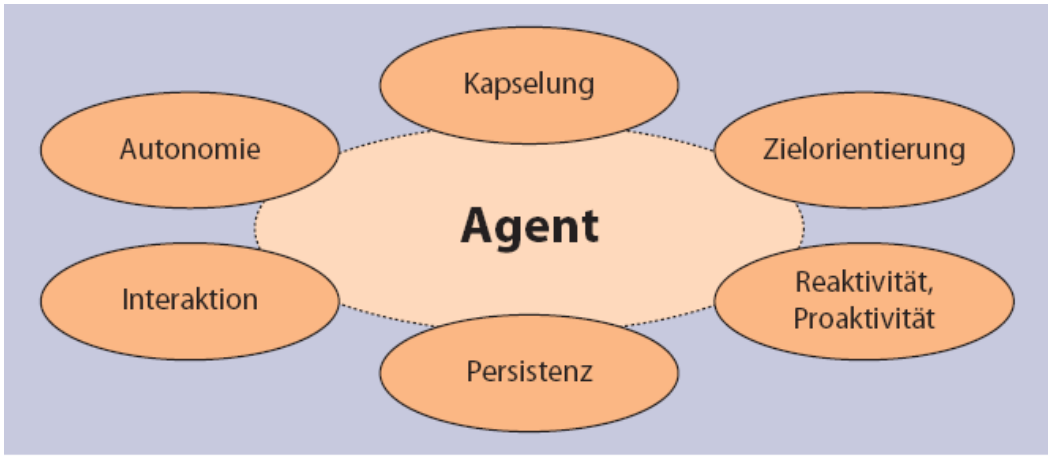
- Es kann auch die Abwesenheit von Ereignissen in einem bestimmten Zeitraum ausgewertet werden.

Zustandsorientierte Programmierung

- Zustandsorientierte Programmierungen erlaubt es Funktionalität für Zustandsänderungen anzugringen.
- Zustände stehen damit im Mittelpunkt.

Agentenorientierte Programmierung

- Grundkonzepte:



Agentenorientierte Programmierung

- (Wagner, 2003):

> Im Kontext der agentenorientierten Softwareentwicklung versteht man unter einem Agenten eine abgrenzbare Softwareeinheit mit einem definierten Ziel. Ein Agent versucht, dieses Ziel durch autonomes Verhalten zu erreichen und interagiert dabei kontinuierlich mit seiner Umgebung und anderen Agenten.

- Agentenorientierte Programmierung unterstützt Proaktivität.
- Die Anwendung reagiert damit nicht nur auf Aufruf, sondern kommuniziert zeitgesteuert mit seiner Umgebung.

Funktionale Programmierung

- Funktionale Programmierung geht von Zustandslosigkeit des Programms aus.
- Damit treten keine Nebeneffekte auf.
- Bekannte Sprachen sind Lisp oder Haskell.

- Beispiel Haskell:

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs

map quadrat [1,2,3] = [quadrat 1, quadrat 2, quadrat 3] = [1,4,9]
```

Logische Programmierung

- In der Logischen Programmierung können Fakten und Regeln definiert werden.
- Abfragen nehmen diese als Grundlage und liefern mittels mathematischer Logik Antworten.
- Logische Programmierung eher deklarativ als imperativ.
- Die bekannteste logische Programmiersprache ist Prolog.
- Beispiel:

```
mann(philip).
mann(peter).
frau(eva).
frau(uta).

?- mann(otto)
yes.
```

```
grossvater(X, Y) :-
   vater(X, Z),
   vater(Z, Y).

vater(philip, peter)
vater(peter, eva)

?- grossvater(philip, eva)
true.
```

Modellgetriebene Programmierung

- (Gruhn, et. al., 2006, S. 186 ff.):

> Hauptaufgabe des Domain Engineering ist die Schaffung und Wartung einer technischen und methodischen Infrastruktur zur Anwendung eines MDA-basierten Ansatzes (bzw. ein MDA-Framework), während im Application Engineering die Benutzung eben dieses Frameworks zur Erstellung von (Software-) Systemen im Vordergrund steht.

- Typisch für MDAs sind maßgeschneiderte Skriptsprachen, sog. Domain-specific Languages (DSLs) und Codegeneratoren.
- Unterschieden werden verschiedene Modeltypen wie Computation-Independent-Model (CIM), Platform-Independent-Model (PIM), Platform-Specific-Model (PSM), welche sukzessive in Code überführt werden.

Strukturelle Aspekte

- Paradigmen können sich eher ergänzen (z.B. OOP und AOP) oder gelten eher als Alternativen (z.B. OOP und FP).

- Insb. für letzteres und insb. in Multiparadigmensprachen wie C# und Java, die primär objektorientiert sind, aber auch funktionale Programmierung unterstützen, ist es Aufgabe der Architektur, festzulegen, wo z.B. überwiegend funktional programmiert wird.

Literatur

- Buschmann, F. et al., 1996. *Pattern-Oriented Software Architecture, Vol. 1: A System of Patterns.* s.l.:Wiley, West Sussex, England.
- Freeman, E. et. al., 2015. *Entwurfsmuster von Kopf bis Fuß*. O'Reilly.
- Gamma, E., Helm, R., Johnson, R. & Vlissides, J., 1995. *Design Patterns.* Boston: Addison-Wesley.
- Goll, J., 2014. *Architektur- und Entwurfsmuster der Softwaretechnik, 2. Auflage*.
- Gruhn, V., Pieper, D. & Röttgers, C., 2006. *MDA.* Berlin, Heidelberg: Springer.
- Harrison, W. & Ossher, H., 1993. Subject-Oriented Programming: A Critique of Pure Objects. *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications (OOPSLA ´93)*, 28(10), pp. 411-428.

Literatur

- Kiczales, G. et al., 1997. Aspect-Oriented Programming. *ECOOP'97 — Object-Oriented Programming*, pp. 220-242.
- Luckham, D., 2002. *The Power of Events*. s.l.:Addison-Wesley.
- OASIS, 2006. *Reference Model for Service Oriented Architecture, Version 1.0.* [Online] Available at: http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf
- Szyperski, C., 1996. Definition of a software component. In First International Workshop on Component-Oriented Programming (WCOP'96), *ECOOP'96*, Linz, Austria.