1 Objektorientiertes Design

1.1 SOLID

Single Responsibility Jede Klasse sollte nur eine Verantwortung haben (Unix-Philosophie)

Grund: steigende Änderungswahrscheinlichkeit führt zu steigender Fehlerwahrscheinlichkeit

Open-Closed-Prinzip Klassen sollten für Erweiterungen offen sein, gleichzeitig aber ihr Verhalten nicht ändern. (open for extension, closed for modification)

Beispiel: Vererbung ändert Basisverhalten nicht, erweitert aber um zusätzliche Funktionen oder Daten.

Liskovsches Substitutionsprinzip Instanzen von abgeleiteten Klassen müssen sich so verhalten, dass sie das Verhalten der Basisklasse 1:1 abbilden können. Überschriebene Methoden dürfen ihr Basisverhalten nicht ändern.

Ziel: Vermeidung von Überraschungen beim Anwender.

Interface Segregation Clients sollten nicht dazu gezwungen werden, von Interfaces abzuhängen, die sie nicht verwenden. Clients agieren nur mit Interfaces, die das und *nur* das implementieren, was die Clients benötigen.

Ziel: Aufteilung großer Interfaces und Reduktion unnötiger Abhängigkeiten.

Dependency Inversion Abhängigkeiten sollten von kronkreteren Modulen niedrigerer Ebenen zu abstrakten Modulen höherer Ebenen gerichtet sein.

Ziel: Reduktion von Abhängigkeiten zwischen Modulen.

Dependency Lookup Objekt sucht nach benötigtem Objekt, beispielsweise in einem Register. Damit hängt es nicht von jedem benötigten Objekt ab, sondern nur vom Register. Das Objekt stellt zur Laufzeit seine Verknüpfung mit einem anderen Objekt her, indem es nach einem logischen Namen sucht. (Aktiv)

Beispiel: Person ruft Escort-Service an und fragt gezielt nach “Scarlett O’Hara”.

Dependency Injection Erzeugung von Objekten und Zuordnungen von Abhängigkeiten werden an eine dritte Partei delegiert, damit sind die Objekte voneinander nicht abhängig. Abhängigkeiten werden von außen injiziert. (Passiv)

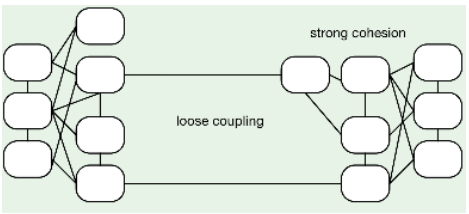
Beispiel: Der Escort-Service weiß, was seine Kunden brauchen und teilt diesen ohne Nachfragen oder Auftrag eine Escort-Dame zu.

1.2 Kopplung

Strong cohesion Eine Klasse mit “strong cohesion” hat wenige, wohldefinierte Aufgaben, die eng miteinander verwandt sind.

Beispiel: High Cohesion: Escort-Dame. Low Cohesion: Partnerin.

Loose coupling Komponenten einer Software kommunizieren nur über wenige Schnittstellen mit anderen Komponenten, dadurch ist die Abhängigkeit sehr gering. Globale Variablen, öffentliche Attribute, Singletons, Zustände in Datenbanken: Schlecht, da sie Schnittstellen vergrößern.



2 Architekturmuster

2.1 Unterschied Architektur- zu Entwurfsmuster

Architekturmuster beziehen sich auf das Gesamtsystem (grundlegendes Problem), während sich Entwurfsmuster auf die Teilgebiete und die Umsetzung beziehen (lokales Problem)

Beispiel: Einkommensgenerierung. Architektur: Gründung Escort-Service. Entwurf: Standort, Personal, Verwaltung.

2.2 Layer

Software ist in logischen Schichten organisiert. Problem:

- Unterteilung komplexer Systeme (Abhängigkeitsminimierung)
- Horizontale Strukturierung (Obere Schicht = Client, Untere Schicht = Server)
- Jede Schicht repräsentiert eine Abstraktion

Lösung:

- Jede Schicht verfolgt einen bestimmten Zweck
- Client-Server Modell

strict layering / closed architecture Schichten dürfen nicht übersprungen werden und dürfen nur die direkt darunter liegende Schicht aufrufen.

Vorteil: leichter test- und wartbar.

loose layering / open architecture Schichten dürfen übersprungen werden, höchste Schicht kann direkt mit tiefster Schicht kommunizieren.

Vorteil: höhere Performanz.

2.3 Pipes and Filters

Filter modifizieren Daten, Pipes transportieren Daten. Problem:

- Unterteilung Datenstrom in Einzelschritte
- Einfach erweiterbar
- Parallelverarbeitung gut möglich

Lösung:

- Zerlegung in Einzelschritte
- Jeder Verarbeitungsschritt ist ein Filter
- Jeder Transportweg ist eine Pipe
- Filter lesen, verarbeiten und geben Daten aus
- Pipes transportieren und puffern Daten
- Pipes entkoppeln Entitäten
- Asynchrone Schreib- und Lesevorgänge möglich

Aktiver Filter Eigenständig laufende Prozesse oder Threads, werden nach Instanziierung von übergeordnetem Programmteil aufgerufen. Laufen kontinuierlich.

Passiver Filter Aufruf durch benachbartes Filterelement.

Push Vorgänger hat Ausgabedaten erzeugt und ruft zu aktivierenden Filter auf. (Daten werden zur Verfügung gestellt)

Pull Nachfolger fordert Daten an. (Daten werden benötigt)

2.4 Plug-In

Interface beschreibt Funktion, Plug-Ins sind die konkreten Umsetzungen dieser Interfaces. Der Plugin-Manager sorgt für die korrekte Zuordnung der konkreten Umsetzung zum Interface bei Aufruf.

Problem:

- Erweiterbarkeit ohne Modifikation
- Neue Funktionalität durch Dritte
- Core-System funktioniert ohne Zusätze, bietet mit ihnen aber mehr Funktionalität

Lösung:

- Interfaces für Erweiterungen
- Plug-Ins implementieren diese
- Plug-Ins ersetzen zur Laufzeit die Interfaces durch konkrete Implementierungen.

Beispiel: Interface: Escort-Dame, Konkrete Umsetzungen: Cindy, Mandy, Scarlett, Plugin-Manager: Telefonistin der Agentur.

2.5 Broker

Broker ist verantwortlich für die Koordination von Kommunikation (Weitergabe von Anfragen und Antworten)

Problem:

- Skalierbarkeit
- Verteilbarkeit
- Lose Kopplung
- Gegenseitige Abhängigkeit nicht sinnvoll

Lösung:

- Komponenten klassifiziert durch deren Rolle
- Serverkomponenten bieten n>=1 Services
- Clients nutzen m>=1 Services
- Rollen können sich ändern
- Broker bietet Vermittlungs- und Zwischenschicht

marshalling Serialisierung von Methodenaufrufen für Netzwerktransmission

unmarshalling Rückkonvertierung in Standard-Methodenaufrufe auf Empfängerseite

Beispiel: Pornokino mit Gloryhole. Die Wand mit dem Loch ist der Broker, die Kundschaft sind die Clients, die Dienstleisterinnen sind die für den Client unzugänglichen Systeme und unterscheiden und/oder ergänzen sich im Funktionsumfang.

2.6 Service-Oriented Architecture (SOA)

Bei einer SOA stellt eine Anwendung Services für andere Anwendungen über eine Schnittstelle, typischerweise über ein Netzwerk, bereit.
Beispiel: Kunde bestellt bei einem Versandhändler.
Erfassung – Verfügbarkeitsprüfung – Bonitätsprüfung – Bestellung – Kommissionierung – Versand – Rechnungsstellung – Zahlungseingang

Für jeden Geschäftsprozess gibt es einen Dienst, diese sind völlig unabhängig voneinander und kommunizieren nur durch das Resultat des Vorgängers mit diesem.

Anforderungen an Services

- Verteilung - Ein Dienst steht in einem Netzwerk zur Verfügung
- Geschlossenheit - Jeder Dienst stellt abgeschlossene Einheit dar, die unabhängig aufgerufen werden kann.
- Zustandslos - Service startet bei jedem Aufruf im gleichen Zustand (keine Informationen eines früheren Aufrufs weitergeben)
- Lose Kopplung - Services sind untereinander entkoppelt. Ein beliebiger Service kann bei Bedarf dynamisch zur Laufzeit vom Servicenutzer gesucht und aufgerufen werden.
- Austauschbarkeit - standardisierte Schnittstelle
- Ortstransparenz - für Nutzer unwichtig auf welchem Rechner Service läuft
- Plattformunabhängigkeit
- Zugriff auf einen Dienst über Schnittstellen - Schnittstellenkentniss reicht zur Nutzung. Implementierung verborgen
- Ein Dienst ist in einem Verzeichnis registriert

2.7 Model-View-Controller (MVC)

Problem: Mehrere Benutzeroberflächen, die sich schnell ändern können. Dargestellte Informationen müssen stets aktuell sein.

Lösung: Aufgabenverteilung auf 3 Komponenten, Model verwaltet und verarbeitet Daten, View stellt Daten dar, Controller verarbeitet Benutzereingaben

Push Model sendet aktiv Daten an die View, welche dann die Darstellung aktualisiert.

Pull View wird durch Model informiert, dass neue Daten vorhanden sind. Wird die View *immer* informiert ist es aktiver Pull, sonst passiver. Der Controller wertet Benutzereingaben aus und aktualisiert ggf. das Model oder die View. Die View zieht sich die neuen Daten nach Information durch das Model.

3 Entwurfsmuster

3.1 Zweck

Wiederverwendbare Vorlage zur Problemlösung, die in einem bestimmten Zusammenhang einsetzbar ist.

3.2 Abstraktion

Abstrakte Klasse Wenn Methoden auf eine bestimmte Art implementiert werden müssen oder wenn non-public Daten oder Methoden enthalten sein sollen. Dies beeinträchtigt nicht die Möglichkeit, reine Methodenköpfe zu definieren, die überschrieben werden müssen.

Interface Reine Schnittstellenbeschreibung.

3.3 Erzeugungsmuster

Abstract Factory / Abstrakte Fabrik Schnittstelle zur Erzeugung einer Familie von Objekten, wobei die konkreten Klassen der zu instanziierenden Objekte nicht näher festgelegt werden.

Factory / Fabrik Objekt, das durch Methodenaufruf andere Objekte erzeugt. Das zurückgegebene Objekt ist neu und verweist nicht zwingend auf die Factory.

Object Pool Objekte, die mehrfach verwendet werden sollen, werden nur bei der ersten Verwendung instanziiert und dann im Objektpool aufgehoben um sie wiederverwenden zu können.
Vorteil: Reduzierter Aufwand (Zeit / Rechenleistung)
Nachteil: Erhöht Komplexität

Singleton Stellt sicher, dass von einem Objekt nur eine Instanz existiert, ist in der Regel global verfügbar.

3.4 Strukturmuster

Adapter Übersetzt eine Schnittstelle in eine andere, dadurch wird die Kommunikation von Klassen mit inkompatiblen Schnittstellen untereinander ermöglicht.

Bridge Dient zur Trennung von Implementierung und ihrer Abstraktion. So können beide unabhängig voneinander verändert werden.

Composition Repräsentiert Teil-Ganzes-Hierarchien, indem Objekte zu Baumstrukturen zusammengefasst werden. Fasst in abstrakter Klasse sowohl primitive Objekte als auch deren Behälter zusammen.

Decorator Flexible Alternative zur Unterklassenbildung um eine Klasse nachträglich um zusätzliche Funktionalitäten zu ergänzen.

Facade Bietet einheitliche, meist vereinfachte Schnittstelle zu einer Menge von Schnittstellen eines Subsystems.

Proxy Überträgt die Steuerung eines Objekts auf ein vorgelagertest Stellvertreterobjekt.

3.5 Verhaltensmuster

Command Kommandoobjekt kapselt einen Befehl um so zu ermöglichen, Objekte in eine Warteschlange zu stellen, Logbucheinträge zu führen und Operationen rückgängig zu machen.

Iterator Stellt die Möglichkeit zur Verfügung, auf Elemente einer aggregierten Struktur sequenziell zuzugreifen, ohne die Struktur zu enthüllen. Auch als Cursor bekannt.

Mediator Dienst zum Steuern des kooperativen Verhaltens von Objekten, wobei Objekte nicht direkt kooperieren sondern über einen Vermittler.

Observer Beobachtetes Objekt hält eine Liste mit Beobachtern und informiert diese über Veränderungen.

State Kapselung zustandsabhängiger Verhaltensweisen von Objekten (vgl. Zustandsautomat)

Strategy Definiert eine Familie von Algorithmen, kapselt diese und macht sie untereinander austauschbar. Lässt den Algorithmus unabhängig von den nutzenden Clients variieren.

Visitor Repräsentiert eine Operation, die auf Elemente einer Objektstruktur ausgeführt wird. Ermöglicht die Definition einer neuen Operation ohne die Klassen der Elemente zu ändern, auf die es ausgeführt wird.

4 Grafiken

4.1 Architekturmuster

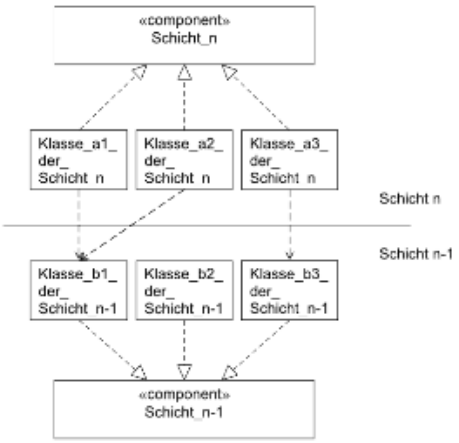


Abbildung 4.1: Layers

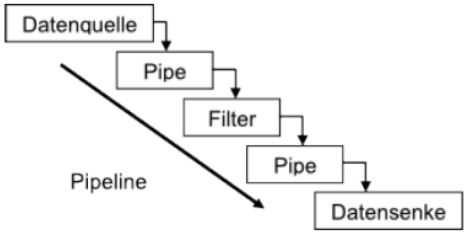


Abbildung 4.2: Pipes and Filters

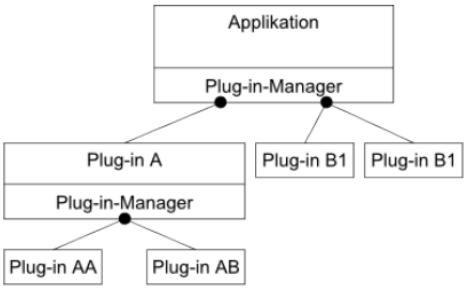


Abbildung 4.3: Plugin

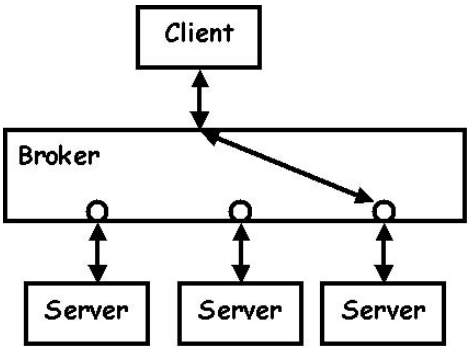


Abbildung 4.4: Broker

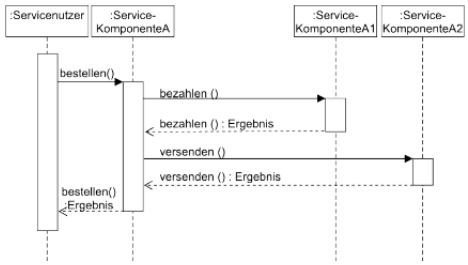


Abbildung 4.5: SOA

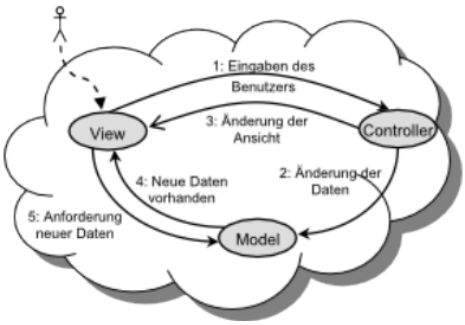


Abbildung 4.6: MVC

4.2 Entwurfsmuster

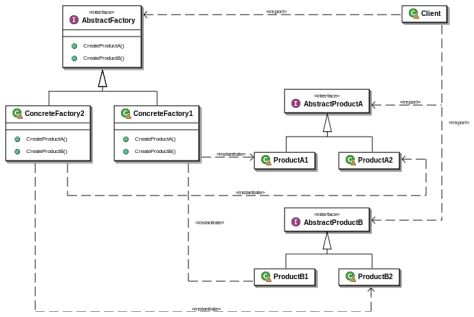


Abbildung 4.7: Abstract Factory

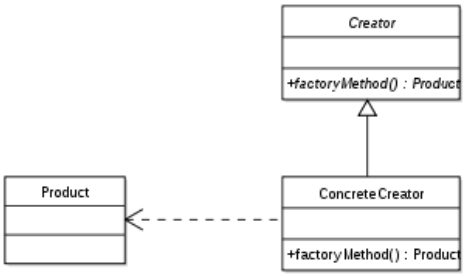


Abbildung 4.8: Factory

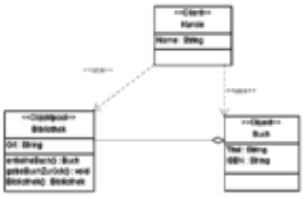


Abbildung 4.9: Object Pool

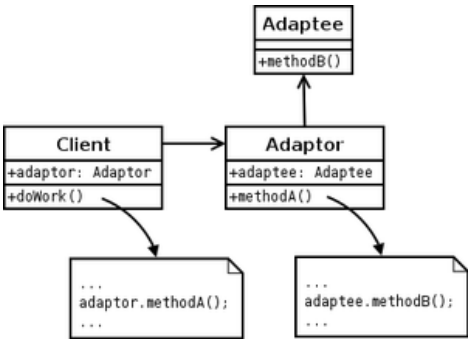


Abbildung 4.10: Adapter

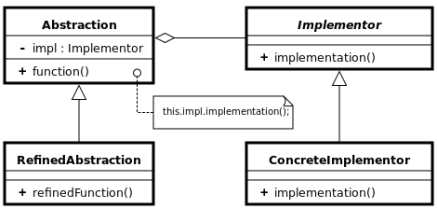


Abbildung 4.11: Bridge

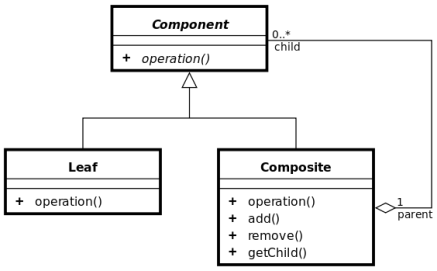


Abbildung 4.12: Composition

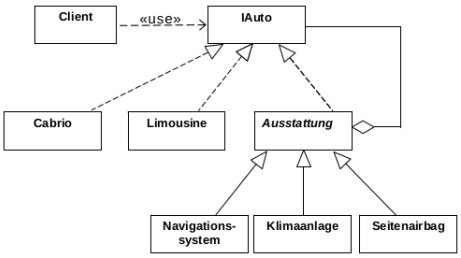


Abbildung 4.13: Decorator

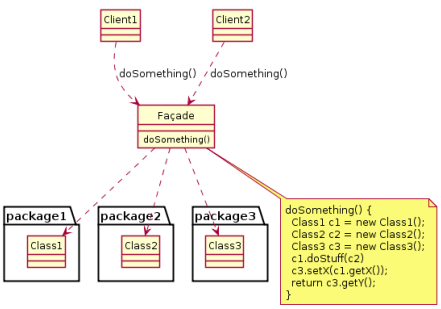


Abbildung 4.14: Facade

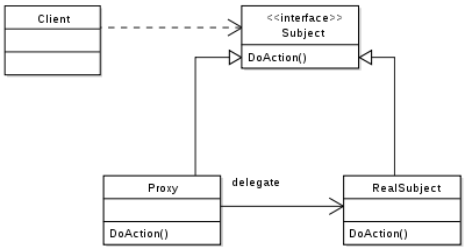


Abbildung 4.15: Proxy

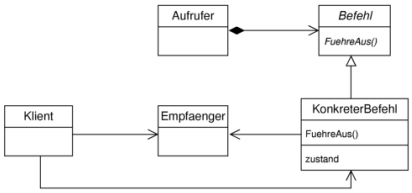


Abbildung 4.16: Command

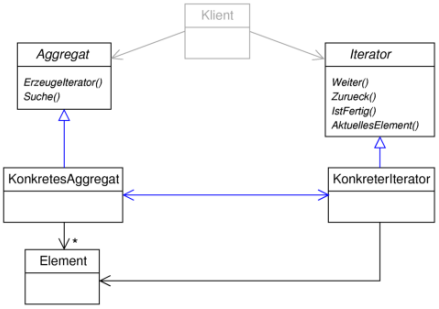


Abbildung 4.17: Iterator

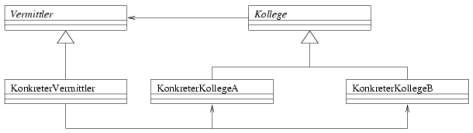


Abbildung 4.18: Mediator

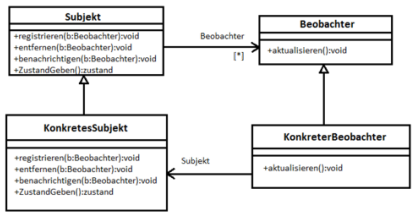


Abbildung 4.19: Observer

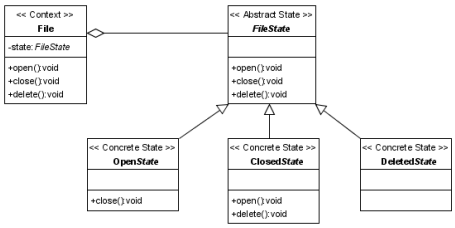


Abbildung 4.20: State

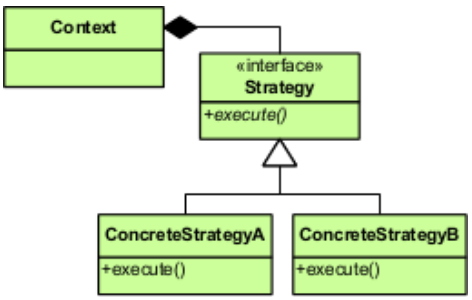


Abbildung 4.21: Strategy

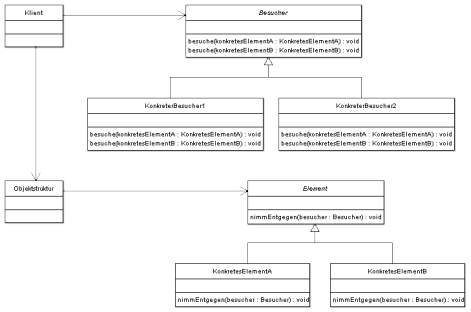


Abbildung 4.22: Visitor