1. LE01 – Einführung und Überblick

Software Engineering

Zielorientierte, systematische Entwicklung von Software anhand eines strukturierten Vorgehens (Plan/Vorgehensmodell), das den Prozess in überschaubare Schritte/Phasen/Meilensteine unterteilt.

Kernprozesse: Anforderungserhebung, Systemdesign, Implementierung, Softwaretest, Softwareeinführung, Wartung/Pflege **Unterstützungsprozesse:** Projektmanagement, Qualitätsmanagement, Risikomanagement

Prozessmodelle im Vergleich

Modell	Vorgehen	Charakter
Hermes (Wasserfall/V-Modell)	Sequenziell, Phasen nacheinander	Definiert, kaum empirisch
(R)UP	Iterativ-inkrementell	Definiert oder empirisch (Tailoring)
Scrum	Iterativ-inkrementell	Empirisch

Iterativ-inkrementell: Software entsteht in Iterationen (z.B. 2-Wochen-Rhythmus). Jede Iteration hat ein Ziel und wird reviewed (Lernprozess, Demming-Cycle). In jeder Iteration: Anforderungen → Analyse & Design → Implementation → Testing.

Vorteil gegenüber Wasserfall: Flexibilität bei unklaren Anforderungen, frühes Feedback, Risiken werden früh erkannt statt erst bei der späten Abnahme.

Die 3 Meilensteine (SWEN1/PM3)

Meilenstein	Bedeutung
M1	Projektskizze
M2	Lösungsarchitektur
M3	Prototyp / Beta-Release

---

2. LE02 – Anforderungsanalyse I

User-Centered Design (UCD)

Anforderungen werden aus Sicht der **Benutzer** erhoben, nicht aus technischer Sicht.

Wichtige Artefakte:

- **Persona –** fiktive, realistische Person die einen typischen Benutzer repräsentiert (Name, Rolle, Ziele, Frustrationen)
- **Kontextszenario –** Prosa-Beschreibung wie eine Persona das System im Alltag nutzt, noch ohne UI/Technik
- **Usage-Szenario –** konkretere Variante mit Bezug zur Lösung

FURPS+ Kategorien

Kürzel	Bedeutung	Beispiel	Typ
F	Functionality	"Login unterstützen"	funktional
U	Usability	"Lernzeit max. 2h"	nicht-funktional
R	Reliability	"Verfügbarkeit 99.9%"	nicht-funktional
P	Performance	"Antwortzeit < 1s", "1000 User gleichzeitig"	nicht-funktional
S	Supportability	"Läuft auf Windows/iOS/Android"	nicht-funktional
+	Constraints	Technologie, Lizenz, Vorgaben	nicht-funktional

Merkhilfe R vs. P: Reliability = Zuverlässigkeit/Verfügbarkeit. Performance = Geschwindigkeit/Durchsatz/Last.

F = funktionale Anforderungen (was das System tut). **URPS+** = nicht-funktionale Anforderungen (wie gut/unter welchen Bedingungen).

---

3. LE03 – Anforderungsanalyse II: Use Cases & SSD

Use Cases – 3 Detailstufen

Stufe	Beschreibung	Wann
Brief	1 Absatz, nur Hauptablauf	Frühe Phase
Casual	Mehrere Absätze, Haupt + Alternativen	Mittlere Phase
Fully Dressed	Vollständige Tabelle, alle Felder	Detailplanung

Fully Dressed UC enthält: Name, Akteur, Vorbedingung, Nachbedingung, Standardablauf (nummeriert), Erweiterungen (Alternativen/Fehler)

Guter Use Case – Tests

- **Boss-Test:** Wenn du dem Boss sagst "ich habe den ganzen Tag nur diesen UC ausgeführt", ist er zufrieden
- **EBP-Test** (Elementary Business Process): Aufgabe einer Person, an einem Ort, zu einer Zeit, als Reaktion auf ein Business Event
- **Size-Test:** Mehr als 1 einzelne Interaktion

Titel: Aktiv formuliert, Verb + Objekt (z.B. "Kasse eröffnen"), beschreibt Ziel des Akteurs.

Use-Case-Diagramm – Beziehungen

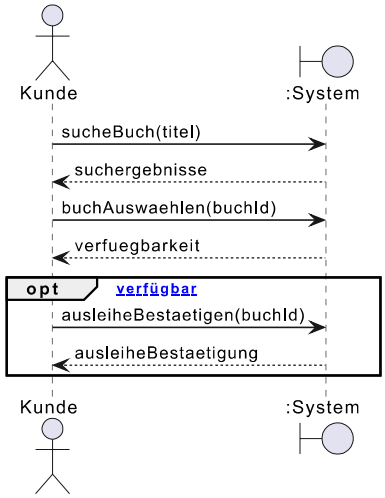
Beziehung	Bedeutung	Pfeilrichtung
Assoziation	Akteur nutzt UC	Akteur → UC
`<<include>>`	UC A ruft UC B **zwingend** auf	A → B
`<<extend>>`	UC B erweitert UC A **optional**, unter Bedingung	B → A
Akteur-Generalisierung	Y erbt von X, kommuniziert mit denselben UCs	Y → X

Merkhilfe: include = "ich brauche dich immer". extend = "ich erweitere dich manchmal".

Systemsequenzdiagramm (SSD)

Zeigt Interaktion Akteur ↔ System als **Black Box** — keine internen Klassen, keine `S -> S` Aufrufe!

- Operationen wie Methodennamen: `sucheBuch(titel)`
- Rückgabewerte als gestrichelte Antwortnachricht
- `opt [Bedingung]` für optionale Blöcke
- `alt [Bedingung A] / else [Bedingung B]` für Alternativen
- `loop [Bedingung]` für Wiederholungen



---

4. LE04 – Domänenmodellierung

Ein **Domänenmodell** ist ein vereinfachtes UML-Klassendiagramm, das die **Fachlichkeit** beschreibt — noch keine Software, nur die reale Welt ("Problemdomäne").

Regeln

- Konzepte (Klassen) in **Einzahl**
- **Keine Methoden**, keine Typen bei Attributen
- Assoziationen mit **Name + Multiplizitäten** an beiden Enden
- Vorgehen wie ein **Kartograf**: bestehende Begriffe verwenden, nichts erfinden
- Bewährte Analyse-Patterns nutzen

Beziehungstypen

Typ	Symbol	Bedeutung	Test
Assoziation	`--`	allgemeine Beziehung	—
Aggregation	`o--`	Teil-Ganzes, Teil existiert unabhängig	"Kann das Teil ohne das Ganze existieren?" → Ja
Komposition	`*--`	Teil-Ganzes, Teil existiert NICHT unabhängig	"Kann das Teil ohne das Ganze existieren?" → Nein
Generalisierung	`<|--`	Spezialisierung/Vererbung	"Ist-ein"-Beziehung

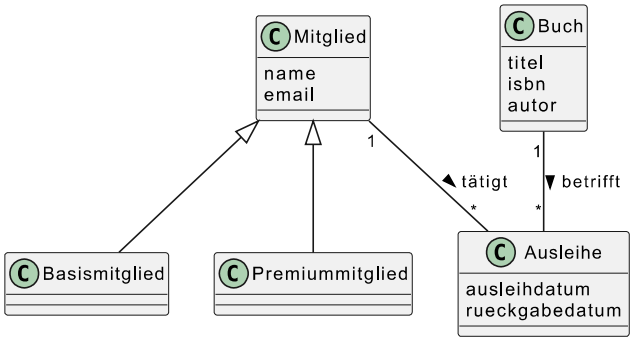
Empfehlung: Aggregation/Komposition nur verwenden, wenn es einen guten Grund gibt — sonst reicht eine einfache Assoziation.

Zustände im Domänenmodell

Zustände nicht über Spezialisierung modellieren (Problem: Zustandswechsel unmöglich ohne Objekt zu ersetzen), sondern über eine **eigene Zustands-Hierarchie** — entspricht dem späteren **State-Pattern** (LE09).

Assoziationsklasse

Wenn eine Beziehung selbst Attribute braucht (z.B. `Ausleihe` zwischen `Mitglied` und `Buch`), wird sie oft als eigene Klasse mit Assoziationen zu beiden Seiten modelliert.



---

5. LE05 – Softwarearchitektur und Design I

Logische Architektur

Aufteilung des Systems in **Schichten (Layers)** und **Pakete**, um Komplexität zu beherrschen.

Schichten-Entwurfsmuster

Schicht	Aufgabe
Präsentationsschicht (UI)	Darstellung, Benutzerinteraktion
Domänenschicht (Business Logic)	Fachlogik, Geschäftsregeln
Datenhaltungsschicht (Persistence)	Speichern/Laden von Daten

Regel: Abhängigkeiten zeigen immer nach **unten**. Obere Schichten kennen untere, nie umgekehrt (Separation of Concerns).

Einflussfaktoren auf Architektur

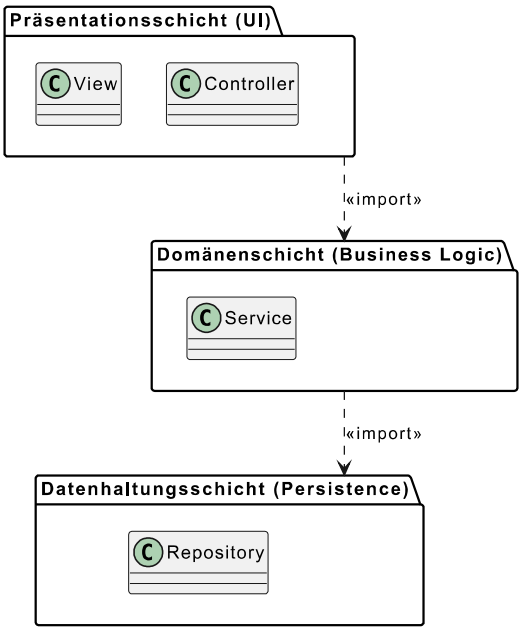
Kommen aus den **nicht-funktionalen Anforderungen** (URPS+): Skalierbarkeit (P), Wartbarkeit (S), Portabilität (+), z.B. "DB-Anbieter austauschbar" → Low Coupling zur Persistenzschicht nötig.

Wichtige Architekturpatterns

Pattern	Beschreibung
Layered Pattern	Strukturierung in Schichten
Client-Server Pattern	Server stellt Services für mehrere Clients bereit
Master-Slave Pattern	Master verteilt Arbeit auf mehrere Slaves
Pipe-Filter Pattern	Verarbeitung eines Datenstroms
Broker Pattern	Meldungsvermittler zwischen Endpunkten
Event-Bus Pattern	Publish/Subscribe über Kanal
MVC Pattern	Model / View / Controller

Paketdiagramm

Zeigt Schichten/Module und ihre Abhängigkeiten (`<<import>>`).



---

6. LE06 – Softwarearchitektur und Design II: GRASP & Klassenentwurf

GRASP – General Responsibility Assignment Software Patterns

Prinzipien (nach Craig Larman) zur sinnvollen Zuweisung von Verantwortlichkeiten an Klassen.

Die 9 Prinzipien

#	Prinzip	Problem/Frage	Lösung
1	**Information Expert**	Wer übernimmt eine Verantwortlichkeit?	Die Klasse, die über die nötigen Informationen verfügt
2	**Creator**	Wer erzeugt ein Objekt B?	Klasse A, wenn A B enthält/aggregiert, registriert, eng zusammenarbeitet, oder Initialisierungsdaten für B hat
3	**Controller**	Wer empfängt System-Events (UI-Schicht)?	Fassaden-Controller (1 Klasse, z.B. "Register") oder Use-Case-Controller (mehrere, pro UC)
4	**Low Coupling**	Wie Abhängigkeiten gering halten?	Variante mit wenigen Kopplungen bevorzugen
5	**High Cohesion**	Wie eine Klasse fokussiert halten?	Verantwortlichkeiten eng & klar abgrenzen
6	**Polymorphism**	Wie typabhängiges Verhalten behandeln?	Verhalten in Unterklassen / via Interface, statt if/else
7	**Pure Fabrication**	Keine passende Domänenklasse vorhanden?	Künstliche Klasse erfinden für gute Kohäsion (z.B. Repository)
8	**Indirection**	Wie direkte Kopplung vermeiden?	Vermittler-Objekt dazwischenschalten
9	**Protected Variations**	Wie vor Änderungen an Variationspunkten schützen?	Stabile Schnittstelle (Interface) um den Variationspunkt

Die 5 wichtigsten für UC-Realisierung: Information Expert, Creator, Controller, Low Coupling, High Cohesion.

Wichtig: GoF-Patterns sind oft **Spezialfälle** von GRASP, z.B.:

> *"Low coupling is a way to achieve protection at a variation point. Polymorphism is a way to achieve protection at a variation point, and a way to achieve low coupling. An indirection is a way to achieve low coupling. The Adapter design pattern is a kind of Indirection and a Pure Fabrication, that uses Polymorphism."*

```
Protected Variation
   └── Low Coupling ──┬── Polymorphism
                      ├── Indirection
                      └── Pure Fabrication
                            └── Adapter (GoF)
```

Controller – Varianten

Typ	Wann
Fassaden-Controller	1 Klasse für alle Systemoperationen (z.B. `Register`). Für kleine Anwendungen.
Use-Case-Controller	Mehrere Controller, je 1 pro UC/UC-Gruppe (z.B. `ProcessSaleHandler`, `HandleReturnsHandler`). Wenn Fassaden-Controller zu gross/inkohäsiv wird.

UML-Klassendiagramm – Notationselemente

Klasse: Name, Attribute (mit Sichtbarkeit `+ - #` und Typ), Operationen

Sichtbarkeiten:

Symbol	Bedeutung
`+`	public
`-`	private
`#`	protected

Beziehungen:

Element	Notation	Bedeutung
Assoziation	`A -- B`	allg. Beziehung, mit Rollen/Multiplizität
Aggregation	`A o-- B`	Teil-Ganzes, lose
Komposition	`A *-- B`	Teil-Ganzes, stark
Generalisierung	`A <|-- B`	B erbt von A
Interface-Realisierung	`A <|.. B`	B implementiert Interface A
Abhängigkeit	`A ..> B`	A nutzt B (z.B. als Parameter), oft `<<use>>`
Abstrakte Klasse	*kursiv* oder `{abstract}`	kann nicht instanziiert werden

Lollipop/Socket-Notation: Alternative Darstellung für Interface-Realisierung (provided) und -Nutzung (required interface).

Interaktionsdiagramme: Sequenz- vs. Kommunikationsdiagramm

Beide zeigen: **Wer tauscht mit wem welche Informationen in welcher Reihenfolge aus?**

Sequenzdiagramm – Notationselemente:

- Lebenslinie, Aktionssequenz
- Synchrone Nachricht `->`, Antwortnachricht `-->` (gestrichelt)
- Gefundene/verlorene Nachricht
- Kombiniertes Fragment (`opt`, `alt`, `loop`)

- Erzeugungs-/Löschereignis (`create`, `destroy`)
- Selbstaufruf, Interaktionsreferenz
- Asynchrone Nachricht (offener Pfeilkopf)

Kommunikationsdiagramm – Notationselemente:

- Lebenslinie (Box), nummerierte Nachrichten (hierarchisch: 1, 1.1, 2, 2.1…)
- Bedingte Nachricht `[Bedingung]`
- Iteration `*[foreach ...]`
- Antwortnachricht mit Rückgabewert `r = m(x,y)`
- Parallele Nachricht `1a/1b`

---

7. LE07 – Use-Case-Realisierung

Was ist eine UC-Realisierung?

Zeigt, wie ein **Szenario eines Use Cases** mithilfe zusammenarbeitender Objekte realisiert wird — klärt, wie aus den UML-Artefakten lauffähiger Code abgeleitet wird.

Relevante UML-Artefakte

- Use Case (das ausgewählte Szenario)
- Systemsequenzdiagramm (Akteur ↔ System)
- Domänenmodell
- Systemoperationen

Vorbereitungsschritte

1. UC + Szenario auswählen
2. Systemoperation auswählen und im Detail definieren
3. Operation Contract erstellen

Vorgehen für die Realisierung (pro Systemoperation)

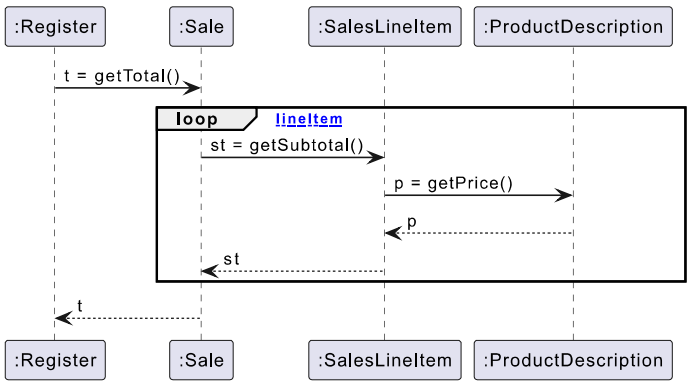
1. **Controller-Klasse bestimmen** (GRASP Controller)
2. **Zu verändernde Klassen festlegen**
3. **Weg zu diesen Klassen festlegen —** Aufruf weiterleiten mit Parametern, Verantwortlichkeiten gemäss **Information Expert** zuweisen, Varianten nach **Low Coupling/High Cohesion** bewerten
4. Veränderungen gemäss Systemvertrag (Operation Contract) programmieren
5. Review bzgl. **High Cohesion** und Architekturkonformität

Beispiel: `makeNewSale()` (NextGenPos)

1. Controller: `Register` ist das System → wird Controller-Klasse
2. Veränderung: neue `Sale`-Instanz erzeugen, als aktuelle Sale in Register ablegen
3. Weg: Register ist bereits Controller, keine Zwischenschritte nötig
4. **Creator-Pattern**: Register ist Container von Sale → Register erzeugt Sale

Beispiel: `getTotal()` (NextGenPos)

1. Controller: bereits `Register`
2. Ziel: Rückgabewert = Gesamtbetrag der aktuellen Sale
3. **Information Expert**: `Sale` kennt seine `SalesLineItem`s → ist Experte
4. `Sale` braucht neue Methode `getTotal()`, benötigt dafür auch `SalesLineItem` und `ProductDescription`



---

8. LE08 – Design Patterns I

Was sind Design Patterns?

Bewährte Lösungsschablonen für wiederkehrende Entwurfsprobleme. Beschreibungsschema: **Name – Problem – Lösung – Hinweise – Beispiele**

Warum? Rad nicht neu erfinden, gemeinsame Sprache, Best Practices lernen.

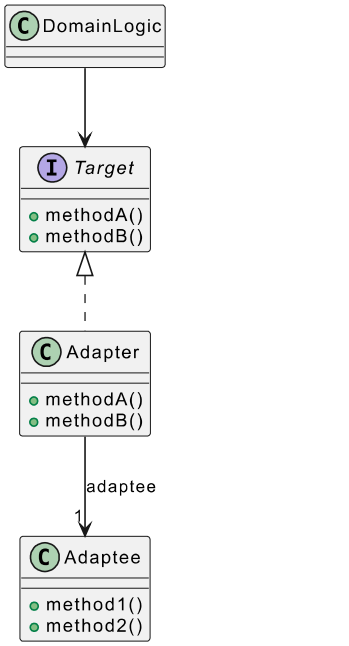
GoF (Gang of Four): Buch "Design Patterns" (1995), 23 Patterns, ca. 15 gebräuchlich. Können als Spezialisierungen von GRASP interpretiert werden.

Patterns dieser LE: Adapter, Simple Factory, Singleton, Dependency Injection, Proxy, Chain of Responsibility

---

Adapter

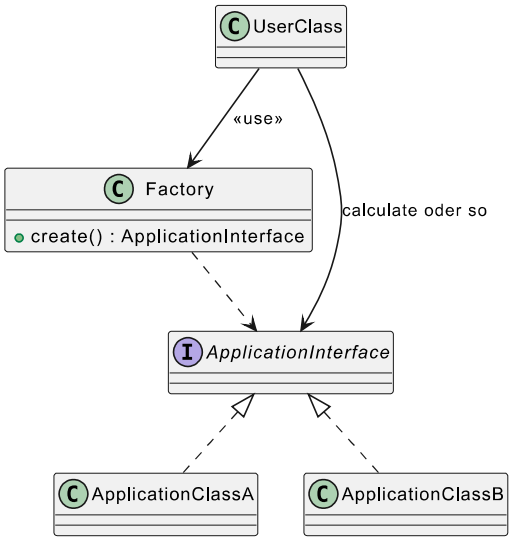
Problem: Eine Klasse soll eingesetzt werden, ist aber inkompatibel mit einem bereits definierten domänenspezifischen Interface. **Lösung:** Eigene Adapter-Klasse dazwischenschalten.



Hinweise: Target-Interface ist für die Domänenlogik optimiert, Adaptee oft extern bezogen. Kann auch `AdapteeAdapter` heissen.

Simple Factory

Problem: Erzeugen eines neuen Objekts ist aufwändig. **Lösung:** Eigene Klasse für die Erzeugung.

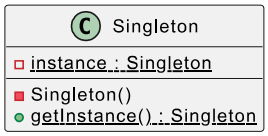


Hinweise: Erzeugung oft konfigurationsabhängig; `create()` kann Parameter haben; Factory kann Objekte cachen/wiederverwenden.

Singleton

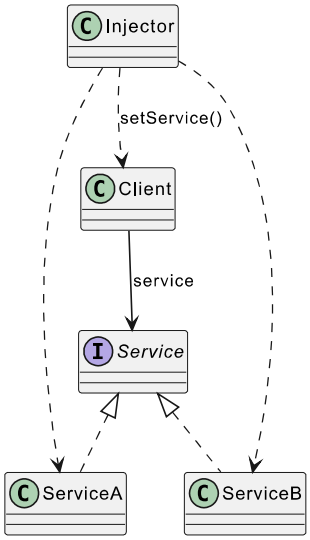
Problem: Nur eine einzige, global sichtbare Instanz einer Klasse nötig. **Lösung:** Klasse mit privatem Konstruktor + statischer `getInstance()`-Methode.

```
public class Singleton {
    private static Singleton instance = new Singleton();
    private Singleton() { }
    public static Singleton getInstance() {
        return instance;
    }
}
```



Dependency Injection (DI)

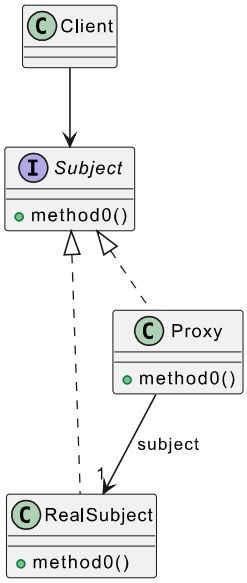
Problem: Klasse braucht Referenz auf ein anderes Objekt (Interface), das je nach Konfiguration unterschiedlich sein kann. **Lösung:** Statt das Objekt selbst zu erzeugen, wird es von aussen (Injector) gesetzt.



Hinweise: Ersatz für Factory-Pattern; **direkter Widerspruch zum GRASP Creator-Prinzip**; Frameworks wie Spring unterstützen DI; erleichtert Testen (Mocks).

Proxy

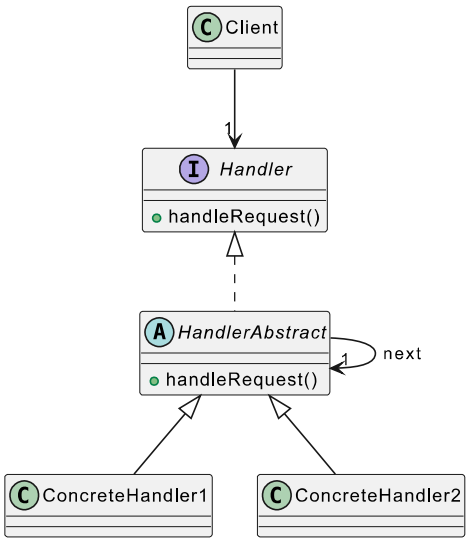
Problem: Zugriff auf ein Objekt soll kontrolliert/vermittelt werden. **Lösung:** Proxy implementiert dasselbe Interface wie RealSubject und delegiert (ggf. mit Zusatzlogik).



Hinweise: Strukturell wie Adapter, aber Adaptee/RealSubject implementiert **dasselbe** Interface. Strukturell identisch mit Decorator (LE09), aber anderer Zweck.

Chain of Responsibility

Problem: Eine Aufgabe kann potenziell von mehreren Handlern übernommen werden, vorab ist nicht klar, welcher zuständig ist. **Lösung:** Handler-Kette, jeder Handler entscheidet ob er bearbeitet oder weiterleitet.



Hinweise: Variante: jeder Handler leitet immer weiter, unabhängig ob er selbst behandelt. Möglich, dass kein Handler zuständig ist.

---

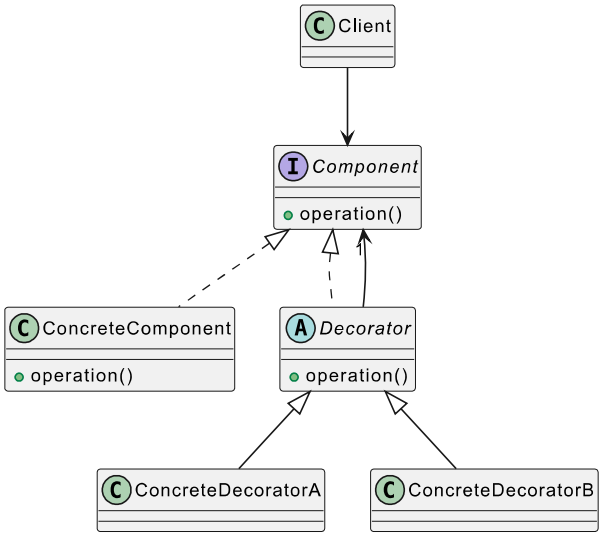
9. LE09 – Design Patterns II

Patterns dieser LE: Decorator, Observer, Strategy, Composite, State, Visitor, Facade

---

Decorator

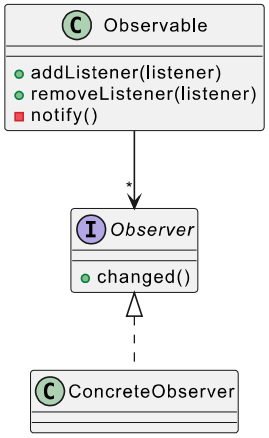
Problem: Ein **Objekt** (nicht die ganze Klasse) soll zusätzliche Verantwortlichkeiten erhalten. **Lösung:** Decorator hat dieselbe Schnittstelle wie das Originalobjekt, wird davor geschaltet, kann Aufrufe selbst behandeln, weiterleiten, oder beides.



Hinweise: Erweitert Funktionalität eines Objekts (im Gegensatz zu Vererbung). Strukturell wie Proxy, aber anderer Zweck. Strukturell wie Composite mit nur 1 Element, aber anderer Zweck.

Observer

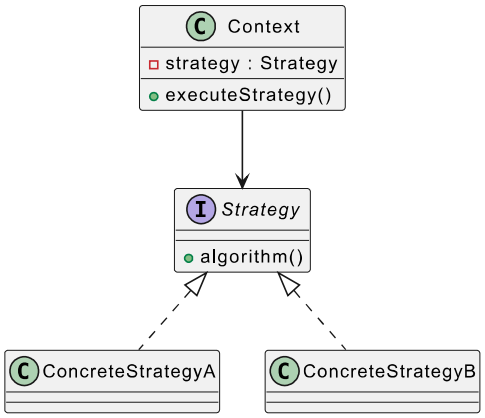
Problem: Ein Objekt soll ein anderes benachrichtigen, ohne dessen genauen Typ zu kennen. **Lösung:** Interface `Observer` definieren; `Observable` (Subject) verwaltet Liste von Observern und benachrichtigt sie bei Änderungen.



Hinweise: Auch "Publish-Subscribe" genannt. Observable kennt nur das Interface, nicht den konkreten Typ → **Low Coupling**. 2 Phasen: Registrierung, dann Benachrichtigung. Oft auch "Listener" genannt; Subject im GoF-Buch.

Strategy

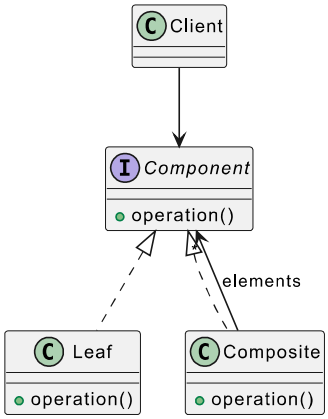
Problem: Ein Algorithmus soll einfach austauschbar sein. **Lösung:** Algorithmus in eigene Klasse mit 1 Methode auslagern, Interface dafür definieren.



Merkhilfe (Abgrenzung zu Template Method): Strategy nutzt **Delegation** für den **ganzen** Algorithmus. Template Method nutzt **Vererbung** für **Teile** des Algorithmus.

Composite

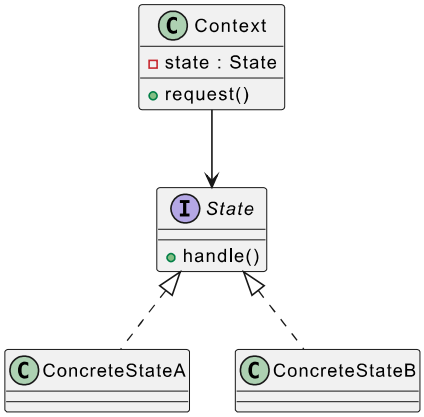
Problem: Objekte sollen einheitlich behandelt werden, egal ob Einzelobjekt (`Leaf`) oder Gruppe (`Composite`) — typisch für hierarchische/Teil-Ganzes-Strukturen. **Lösung:** `Leaf` und `Composite` implementieren dasselbe Interface; `Composite` delegiert Aufrufe an seine Kinder.



Hinweise: Hierarchische Struktur oft vom Fachgebiet vorgegeben. Nicht alle Methoden delegieren 1:1 — Vor-/Nachbearbeitung üblich.

State

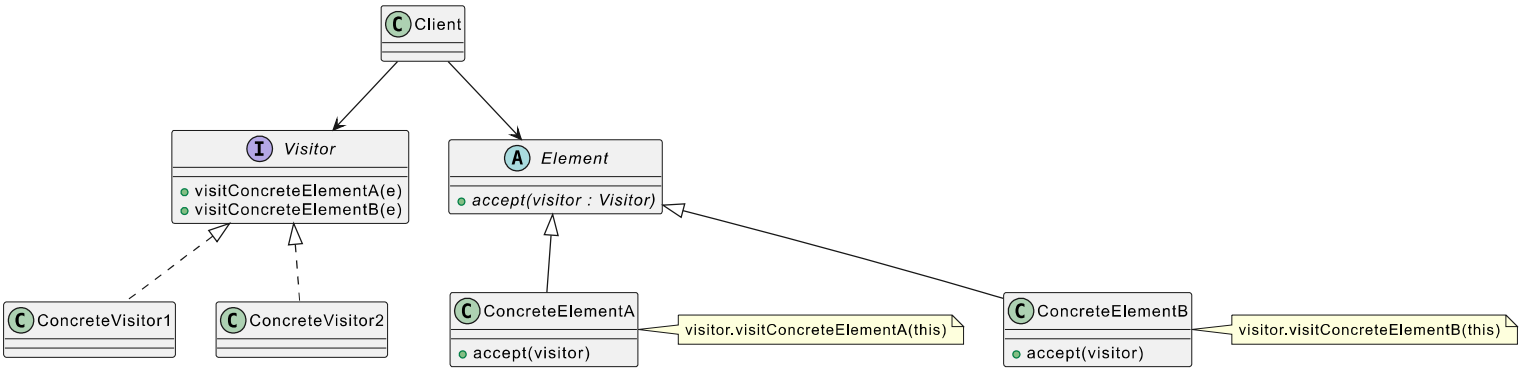
Problem: Verhalten eines Objekts hängt von seinem inneren Zustand ab. **Lösung:** Objekt enthält ein Zustandsobjekt; zustandsabhängige Methoden werden über dieses Zustandsobjekt geleitet.



Bezug zu LE04: Entspricht der "besseren Lösung" für Zustände im Domänenmodell — eigene Zustandshierarchie statt Spezialisierung.

Visitor

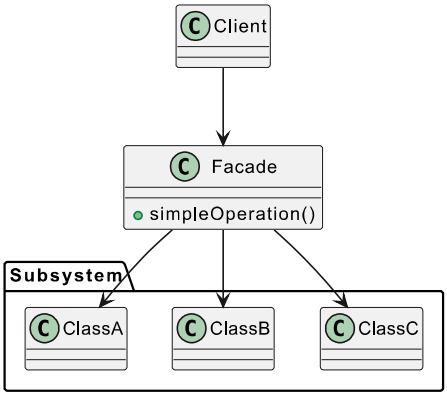
Problem: Klassenhierarchie soll um (eher seltene) Verantwortlichkeiten erweitert werden, ohne viele neue Methoden in jeder Klasse hinzuzufügen. **Lösung:** Visitor-Infrastruktur mit `accept(visitor)` in den Elementen; neue Funktionalität wird in spezifischen Visitor-Klassen realisiert (Double Dispatch).



Hinweise: Widerspricht **Information Expert** (Logik wandert aus der Klasse heraus) — wichtige Methoden trotzdem direkt in der Klasse behalten. Bei mehrstufiger Hierarchie: Frage wer die Kindelemente aufruft.

Facade

Problem: Kompliziertes Subsystem mit vielen Klassen soll für andere Teams einfach nutzbar gemacht werden. **Lösung:** Facade-Klasse bietet vereinfachte Schnittstelle zum Subsystem.



Hinweise: Im Gegensatz zum Adapter kapselt die Facade das Subsystem **nicht vollständig** ab — Parameter/Rückgabewerte dürfen Bezug zum Subsystem haben. Oft vom Framework-Ersteller entwickelt.

---

10. LE10 – Refactoring & Testing

Refactoring

Verbesserung der internen Codestruktur **ohne Änderung des externen Verhaltens**. Ziel: Lesbarkeit, Wartbarkeit, Erweiterbarkeit erhöhen.

Typische Code Smells: Duplizierter Code, lange Methoden, grosse Klassen, lange Parameterlisten, Feature Envy (Methode nutzt mehr Daten einer anderen Klasse als der eigenen).

Typische Refactoring-Techniken: Extract Method, Rename, Move Method, Extract Class, Replace Conditional with Polymorphism.

Code Smell	Beschreibung	Refactoring
Duplicated Code	Gleicher Code an mehreren Stellen	Extract Method
Long Method	Methode macht zu viel	Extract Method
Large Class	Klasse hat zu viele Verantwortlichkeiten	Extract Class
Long Parameter List	Methode hat zu viele Parameter	Introduce Parameter Object
Feature Envy	Methode nutzt mehr Daten einer fremden Klasse als der eigenen	Move Method
Switch Statements	Viele if/else oder switch auf Typen	Replace Conditional with Polymorphism

Testing

Ziel des Softwaretestens: Messen der Qualität des Softwaresystems — definierte Anforderungen dienen als Prüfreferenz, um Fehler aufzudecken (ISTQB: Vorbeugung der Fehlerwirkung im produktiven Betrieb).

TDD (Test Driven Development): Test zuerst schreiben (rot) → Code schreiben bis Test grün → Refactoring (Red-Green-Refactor-Zyklus).

---

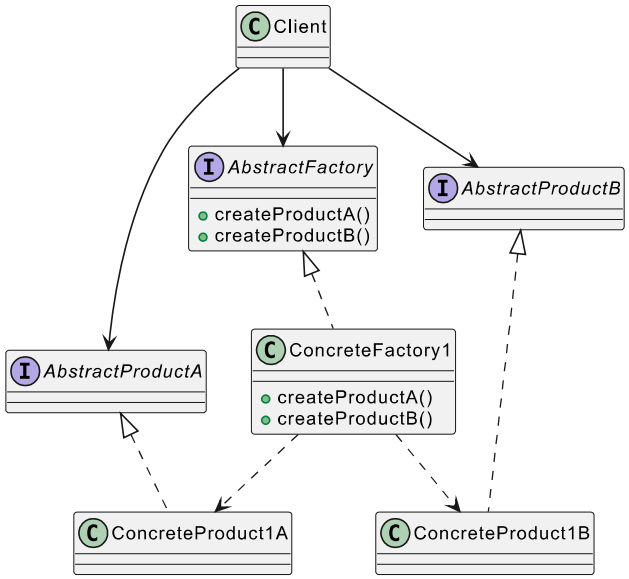
11. LE11 – Design Patterns III (Framework Design)

Patterns dieser LE: Data Access Object, Data Transfer Object, Abstract Factory, Factory Method, Command, Template Method

---

Abstract Factory

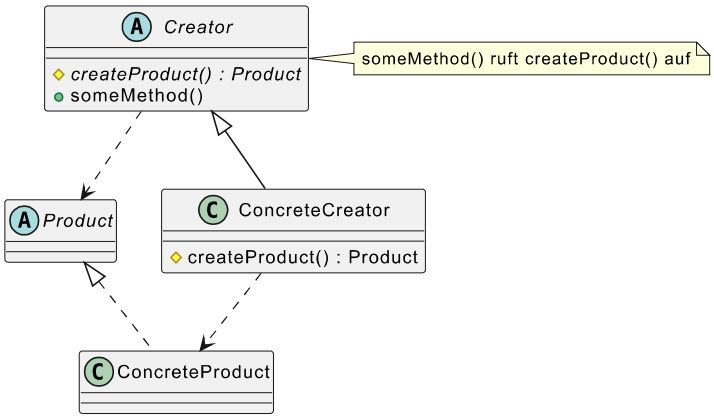
Problem: Erzeugung verschiedener, inhaltlich zusammengehörender Objekte ("Products"), ohne die konkreten Klassen zu kennen — Austauschbarkeit nötig. **Lösung:** `AbstractFactory` mit `create`-Methode pro Produkttyp; konkrete Factories erzeugen konkrete Produktfamilien.



Hinweise: Verallgemeinerung der Simple Factory (LE08). Verschiedene Produkte hängen fachlich zusammen (z.B. Teile derselben Hardware).

Factory Method

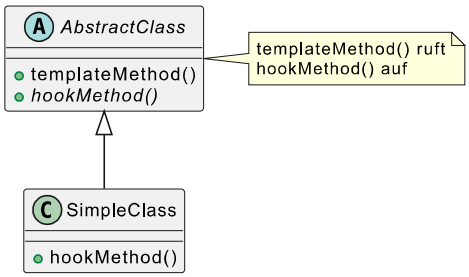
Problem: Eine wiederverwendbare Klasse `Creator` soll eine Instanz von `Product` erzeugen, aber `Product` muss noch spezialisiert werden. **Lösung:** Abstrakte Methode `createProduct()` in `Creator`; konkrete `Creator`-Subklassen erzeugen die passende `Product`-Subklasse.



Hinweise: Parallele Vererbungshierarchien (Creator & Product). Kann als Variante von Template Method interpretiert werden.

Template Method

Problem: Ein Ablauf/Algorithmus soll an bestimmten Punkten anpassbar sein. **Lösung:** Abstrakte Klasse definiert **Template Method** (fertiger Ablauf), die abstrakte **Hook Methods** aufruft. Subklassen überschreiben die Hooks.

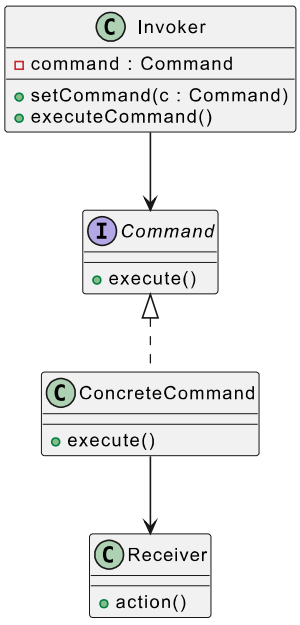


Hollywood-Prinzip: *"Don't call us, we'll call you"* — der Framework-Code ruft den Code der Anwendung auf (nicht umgekehrt).

Hinweise: Hook-Methode kann abstrakt sein oder Default-Implementierung haben. **Verwandtschaft zu Strategy:** Strategy = Delegation für ganzen Algorithmus, Template Method = Vererbung für Teil des Algorithmus.

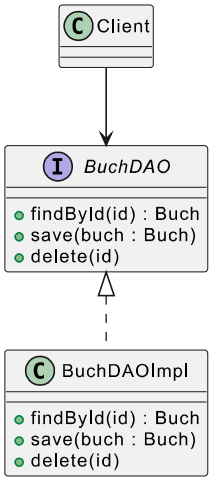
Command

Problem: Eine Aktion/Anfrage soll als Objekt gekapselt werden (für Undo, Queuing, Logging, Parametrisierung von Aufrufern). **Lösung:** `Command`-Interface mit `execute()`; konkrete Commands kapseln Empfänger + Aktion.



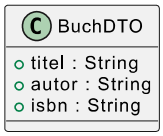
Data Access Object (DAO)

Zweck: Ermöglicht Zugriff auf persistente Daten **unabhängig von der Datenquelle und ihrer Kodierung** (DB, Datei, REST-API, etc.).



Data Transfer Object (DTO)

Zweck: Fasst mehrere Attribute in einem Objekt zusammen, das effizient transportiert werden kann (z.B. über Netzwerk, zwischen Schichten) — enthält i.d.R. keine Geschäftslogik, nur Daten.



Wrap-up: Pattern-Zuordnung nach Zweck

Zweck	Patterns
Objekt-Erzeugung	Simple Factory, Abstract Factory, Factory Method, Singleton, Dependency Injection
Struktur/Kompatibilität	Adapter, Proxy, Decorator, Composite, Facade
Verhalten/Algorithmen	Strategy, Template Method, State, Observer, Visitor, Chain of Responsibility, Command
Datenzugriff	DAO, DTO

---

12. Schnellreferenz UML-Notation (PlantUML)

Notation	Bedeutung	Verwendung
`A <|-- B`	Vererbung (B erbt von A)	Domänenmodell, DCD
`A <|.. B`	Realisierung (B implementiert Interface A)	DCD
`A --> B`	Assoziation / Navigation	DCD, Domänenmodell
`A ..> B`	Abhängigkeit (`<<use>>`)	DCD, Paketdiagramm
`A o-- B`	Aggregation	DCD, Domänenmodell
`A *-- B`	Komposition	DCD, Domänenmodell
`A -> B`	Synchrone Nachricht	Sequenzdiagramm, SSD
`A --> B` (im Sequenzdiagramm)	Antwortnachricht (return)	Sequenzdiagramm, SSD
`A .> B : <<include>>`	Include-Beziehung (zwingend)	Use-Case-Diagramm
`A .> B : <<extend>>`	Extend-Beziehung (optional)	Use-Case-Diagramm
`+` / `-` / `#`	public / private / protected	DCD
kursiv / `{abstract}`	abstrakte Klasse/Methode	DCD

Die wichtigsten Fragen zur Diagrammwahl

Diagramm	Beantwortet die Frage
Use-Case-Diagramm	Wer nutzt welche Funktionen des Systems?
Domänenmodell	Welche fachlichen Begriffe gibt es und wie hängen sie zusammen?
SSD	Wie kommuniziert der Akteur mit dem System (Black Box)?
DCD	Aus welchen Klassen besteht mein System, wie hängen sie zusammen?
Sequenzdiagramm	Wer tauscht mit wem welche Nachrichten in welcher Reihenfolge aus?
Paketdiagramm	Wie ist das System in Schichten/Module gegliedert?

Ende des Cheat Sheets — Viel Erfolg an der SEP! 🎓