Zusammenfassung Entwurf von Softwaresystemen

Wieso Sofware Design?

- Besser für komplexe Anwendungen
  - ↳ Aufteilung in "Teilprobleme"
  - ↳ Bugs sind leuer

Was ist Sofware Design

- Beschreibung von Structure & Behavior
- Sammlung von Objekten (Artifacts), welche die Architektur beschreiben

Modul

→ Programmkomponente die Funktion-/Daten-abstraktion realisiert
  ↳ sind im normalfall nicht alleine stehend ablauffähig

Software komponente

→ konstruktion aus mehreren Modulen mit spezifischen Schnittstellen & Abhängigkeiten
  ↳ können alleine stehend ausgeführt werden.

Sofware System

→ konstruktion aus meheren Software Komponenten mit System- & Nutzer-Schnittstelle.

Design Approches

- **Top-Down**
  ↳ Anforderungen immer weiter runterbrechen
    ↳ Devide & Conquer
- **Bottom-Up**
  ↳ Aufbauend auf basic Elementen und diese zusammenführen zu Sub-Systemen etc.

- **Prozess orientiert** → Fokus auf zu optimierenden Prozess
- **Daten orientiert** → Fokus auf zu automatisierende Daten
- **Objekt orientiert** → Fokus auf Prozess & Daten

General Rule

Low Coupling & High Cohesion

Low Coupling: Dependencies between Modules

High Cohesion: Relationen innerhalb eines Modules

⇒ Wenn ein Modul bearbeitet wird, will man nicht noch andere ändern müssen.

Seperation of Concern (SoC)

→ Fokus auf einer Funktionalität

↳ single responsibility von Klassen, Methoden, Modulen...

Encapsulation

→ Zugriff auf bestimmte Komponenten verhindern

Data Hiding

→ Zugriff auf bestimmte Daten verhindern

YAGNI, KISS, DRY

YAGNI → You arn't gonna need it

↳ implementiere nur, was du brauchst

KISS → Keep it simple & Stupid

↳ einfaches Design = besseres Design

DRY → Don't repeat yourself

↳ Wiederverwendung von Code

UML Diagrams

- Structure Diagram
  - ↳ Package - → organisatioen von Klassen & Dependencies (artifacts)
  - ↳ Class - → Klassen mit ihren Relationen
  - ↳ Component - → Interne Struktur von komponenten

- Behavior Diagram
  - ↳ Activity - → Aktivitätsfluss /workflow
  - ↳ State Machine - → Ablauf verschiedener processe & Events

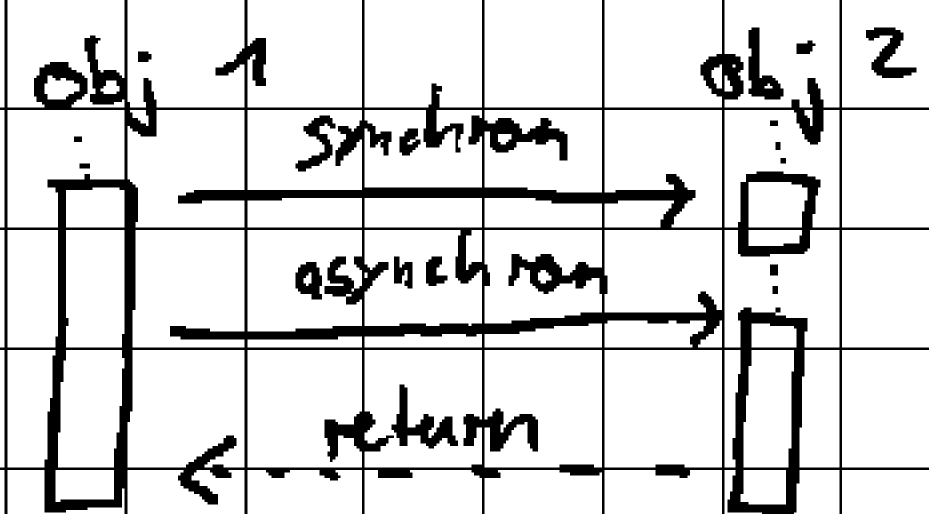
- Interaction Diagram
  - ↳ Sequence - → verfolgt die Ausführung eines Szenarios
  - ↳ Communication - → Interaktions Diagram mit Fokus auf kommunikation zwischen objekten

UML Sequence Diagram

↳ Lifelines of objects

Zeit vergeht an der y-achse

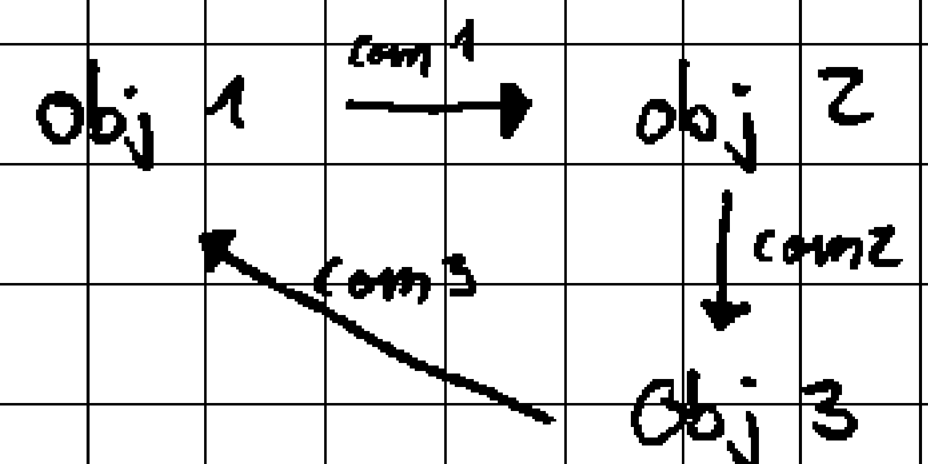
ex:



UML Communication Diagram

↳ zeigt kommunikation zwischen Objekten

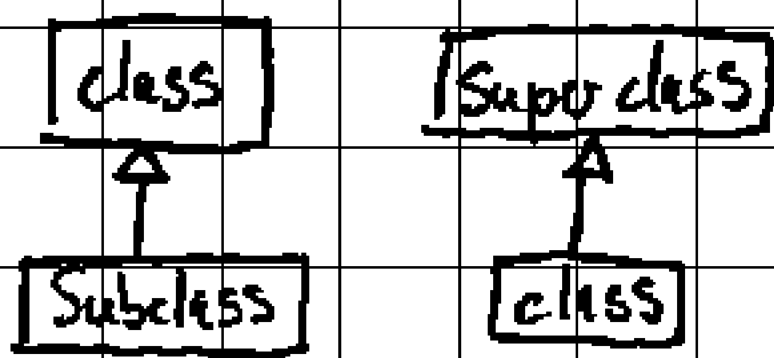
ex:



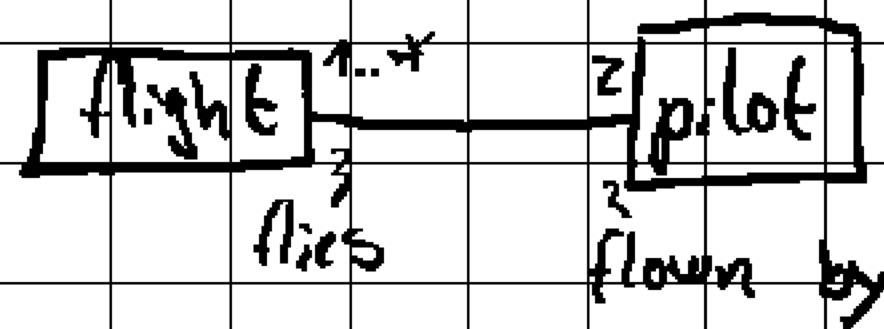
Class Diagram

→ klassen inkl. Methoden & Attributen und ihre Abhängigkeiten

Inhiritence



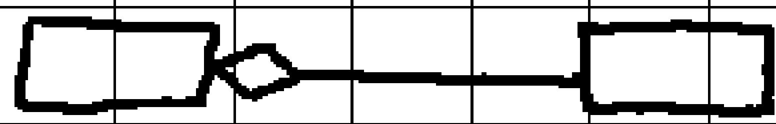
Association



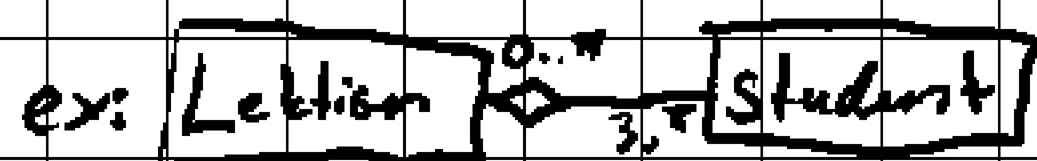
⇒ flight is flown by 2 pilots but 1 pilot can fly 1 - ∞ flights

↳ kann auch directional sein

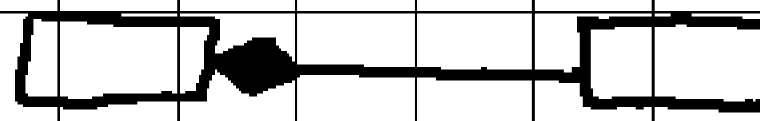
Aggregation



wird diese klasse aufgelöst, bleibt die andere bestehen.



Composition



wird diese klasse aufgelöst löst sich die andere auch auf

UML Cheat Sheet

—▷ Inheritance ----▷ Implementation

methods

name (a: string, b: type): void

⇒ name (name: type): return type
(name: type = variable)

variable

name : type

Visabilities

\+ public − private

\# protected ~ package

static → underlined → <u>variable: string</u>

Abstract class → name kursiv

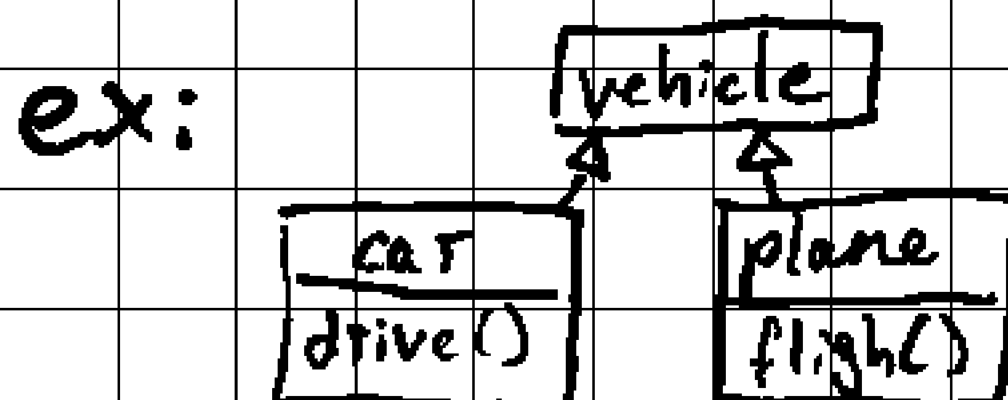
→

Interface → <<interface>>

Lacy Class

↳ eigene klasse für Methoden

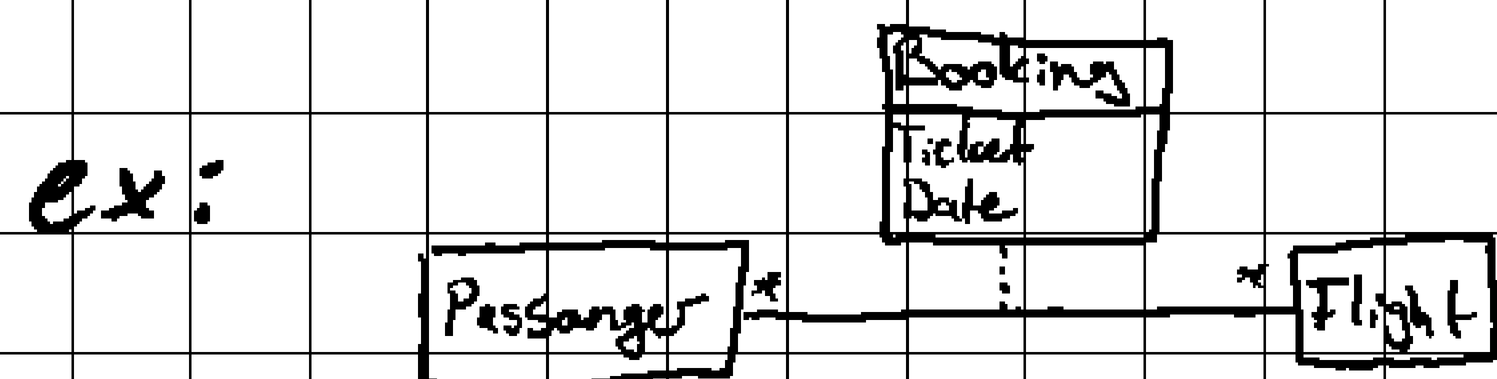
ex:



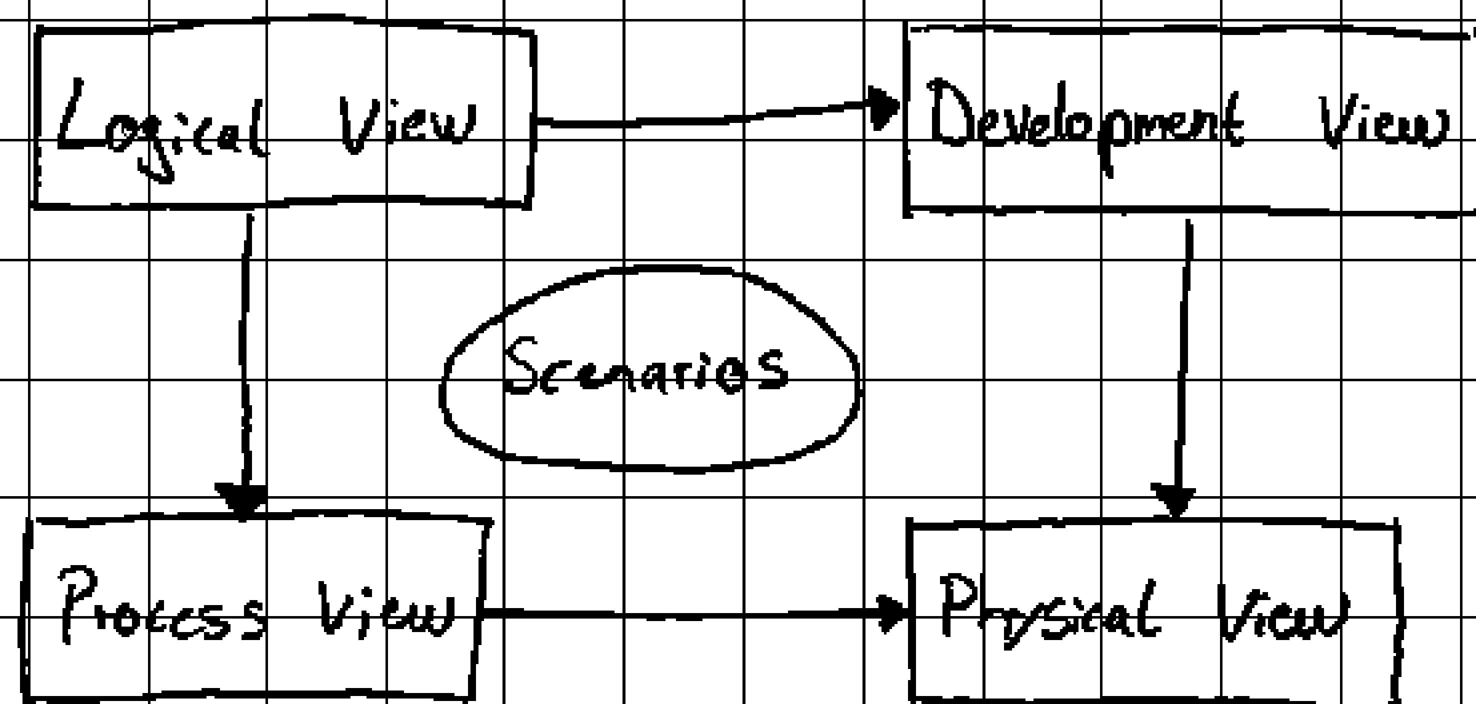
Assocation Class

↳ assoziation mittels 3. klasse

ex:



4+1 View Model



Logical View → architecture

Process View → dynamischer Aspekt von einem System

Development View → Was für klassen, Librarys, Interfaces etc..
(Implementation View)

Physical View → Mapping Software to Hardware

Scenarios → Brings all views together (ex. Use case Diagram) ex. Scenario

Prinzipien in Objekt Orientiertem Design

Information Expert

↳ Verantwortung liegt bei der klasse, die die Informationen hat.

Principle of Creator

klasse B hat die Verantwortung eine Instanz von A zu erstellen wenn

- A teil von B ist
- B hat Instanz von A (aggregation)
- B hat Daten zur Initialisierung von A
- B benützt A

Design Patterns

↳ reusable solution to a recurring problem

↳ wiederverwendbares Lösungsmuster

Types of Patterns

- **Creational Pattern**
  - ↳ helfen bei Erstellung von Objekten
- **Structural Pattern**
  - ↳ helfen dabei verschiedene Teile eines Systems zusammenzupassen
- **Behavioral Pattern**
  - ↳ helfen bei Verwaltung von Algorithmen, Beziehungen, Interaktionen

Singelton Pattern

↳ There can be only one

→ nur eine Instanz einer Klasse mit globalem "point of access"

Merkmale

- Klasse hat private constructor
- Static Methoden
- Static Variabeln

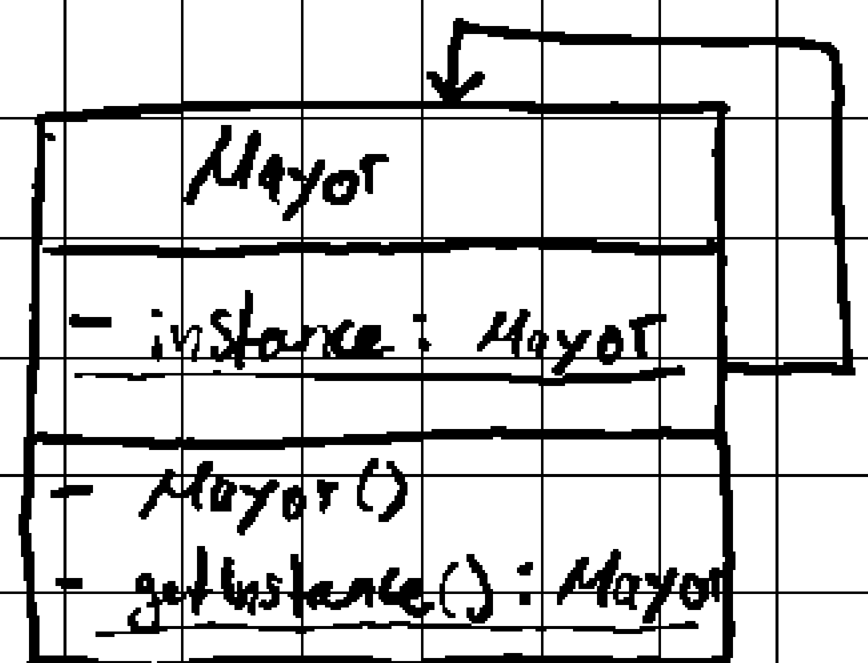
Beispiel

→ eine Stadt hat genau einen Bürgermeister und die ganze Stadt kennt die Telefonnummer, um mit ihm in kontakt zu treten

Java

```
public class Mayor {
  private static Mayor instance;
  private Mayor () {...}
  public static synchronized Mayor getInstance() {
    if (instance == null) instance = new Mayor();
    return instance;
  }
}
```

UML



Factory Pattern

↳ Methode wird genutzt um objekte zu erstellen (nicht der konstruktor direkt)

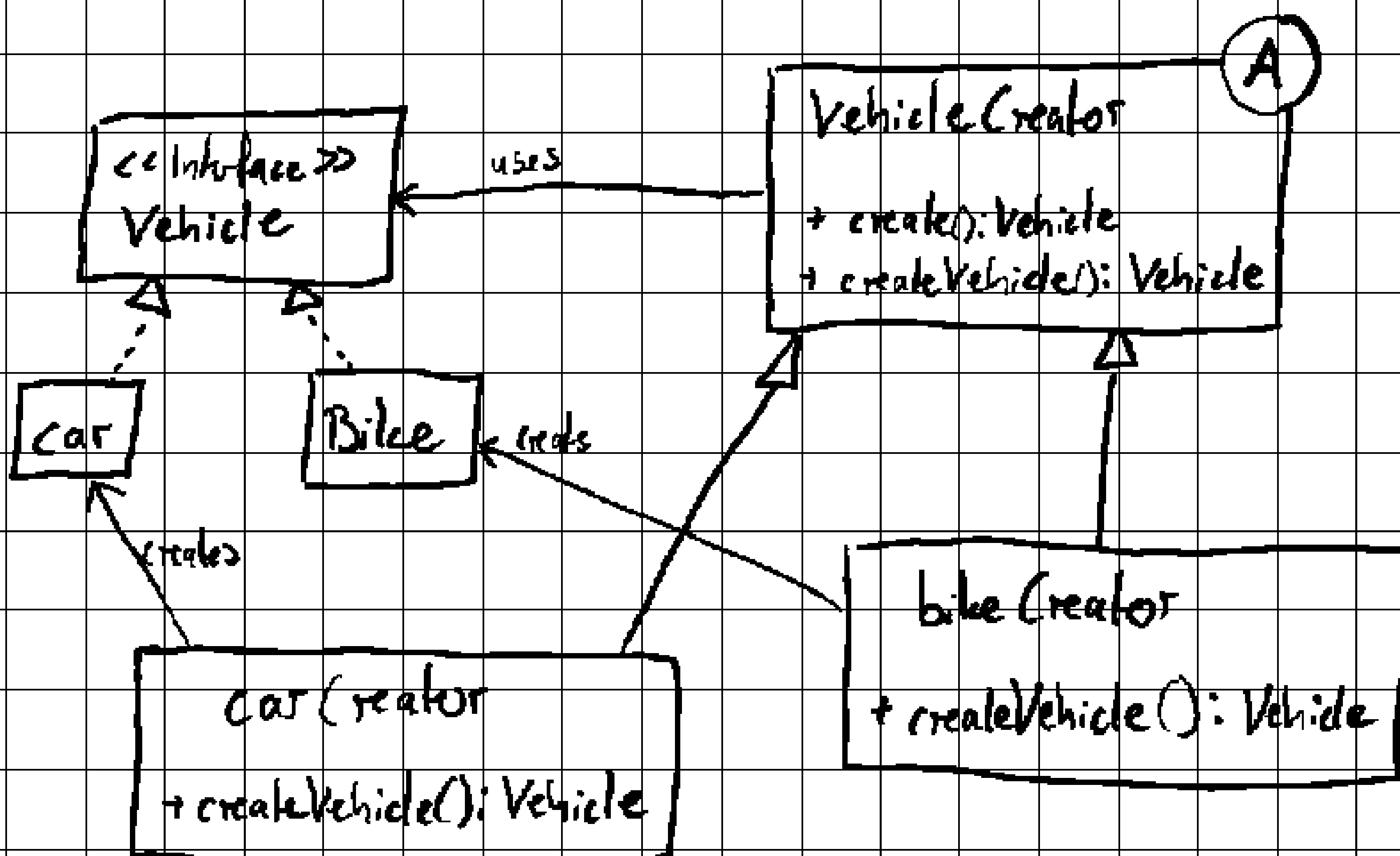
Merkmale

- keine new..() calls, lässt Factory obj erstellen
- kann aus folgenden Teilen bestehen:
  - Product → interface für zu erstellendes Produkt
  - concrete Product → implementierung von Produkt
  - creator → abstrakte klasse für allgemeinen ersteller
  - concrete creator → überschreibt create methode für concrete Prod.

Beispiel

Fabrik die Autos & Motorräder herstellen.

kunde bestellt eines und es wird für ihn angefertigt

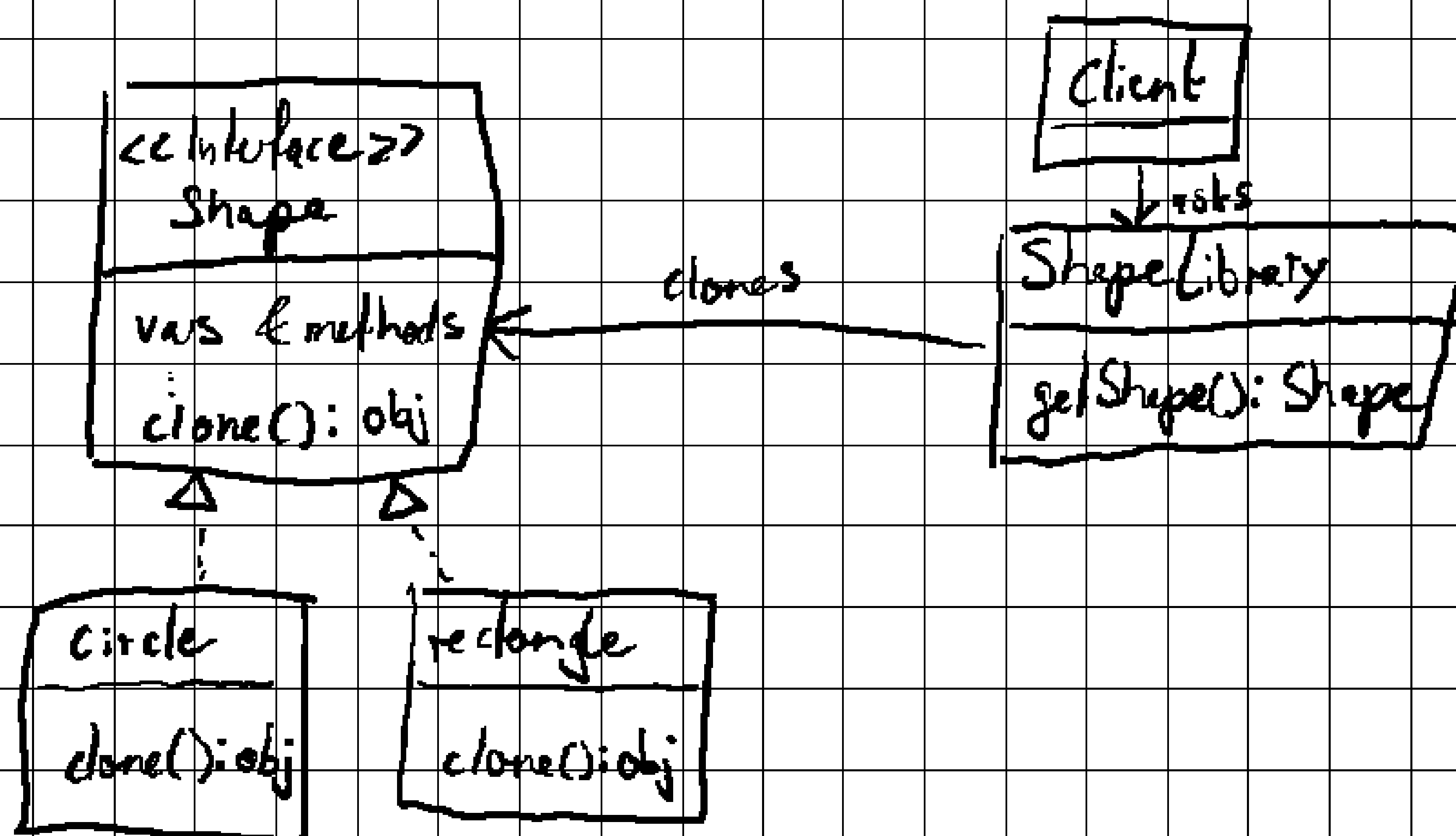


Prototype Pattern

↳ erstellt Template im Hintergrund, kopiert und verändert diesen wenn ein neues Objekt erstellt wird.

Beispiel

↳ Grafik Programm erstellt basic Formen beim Startup und wenn der user ein Neues einfügt wird das template kopiert → spart resourcen.



Composite Pattern

↳ lässt den user einzelne und kollektionen aus mehreren Objekten gleich behandeln

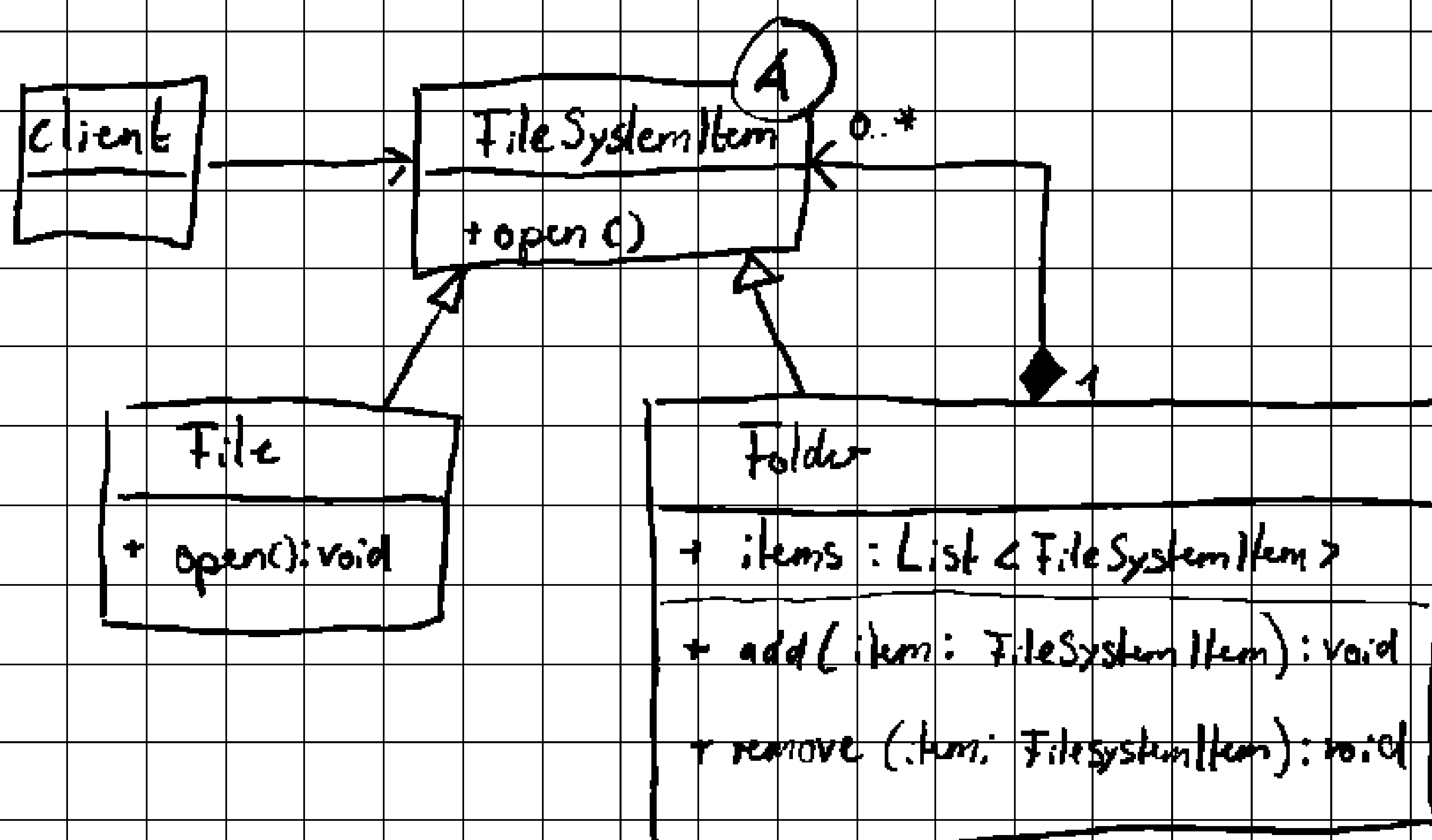
Merkmale

besteht aus folgenden komponenten

- Component → abstraktion aller komponenten mit allgemein nutzbaren Methoden
- Leaf → einzelnes objekt mit dazugehörigen Methoden
- Composite → Liste von Leafs mit Methoden um diese zu manipulieren

Beispiel

↳ Ordner enthält Dateien, oder Unterordner mit Dateien



Decorator Pattern

↳ füge mehr Funktionalität zu einem Objekt hinzu, ohne die Struktur zu ändern

Merkmale

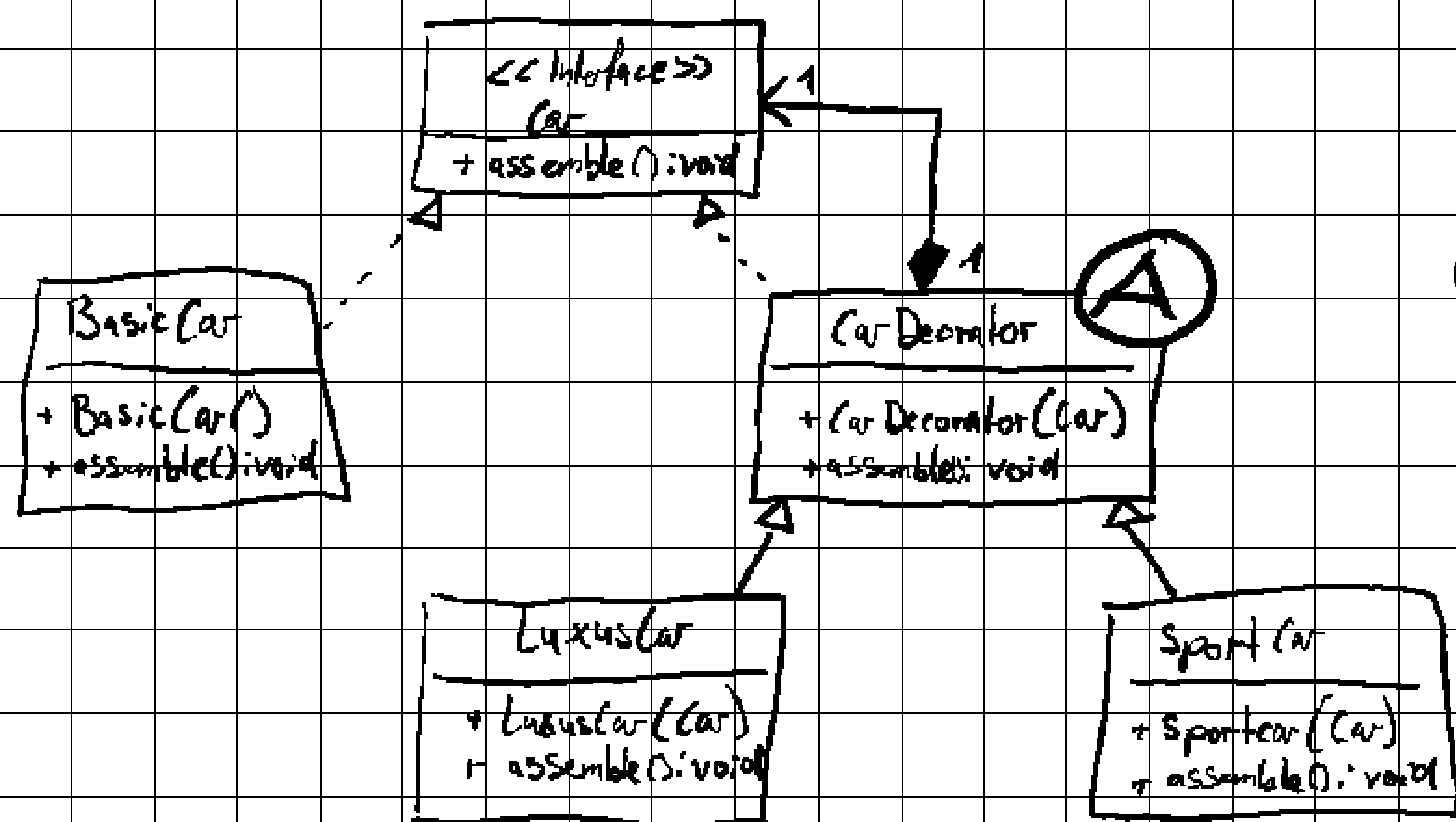
Besteht aus folgenden Komponenten

- Component → allgemeines Interface
- Concrete Component → basic Implementierung
- Decorator → Abstrakte Komponente mit mehr Funktionalität
- concrete Decorator → mehr Funktionalität

Beispiel

↳ Ein Auto kann mit Features von anderen Autos ausgestattet werden.

UML



Java

create Car mit Luxus & Sport Feature

```
Car LuxSport = new LuxusCar(
               new SportsCar(
               new BasicCar()));
```

Adapter Pattern

↳ lässt zwei inkompatible Interfaces zusammen arbeiten

Merkmale

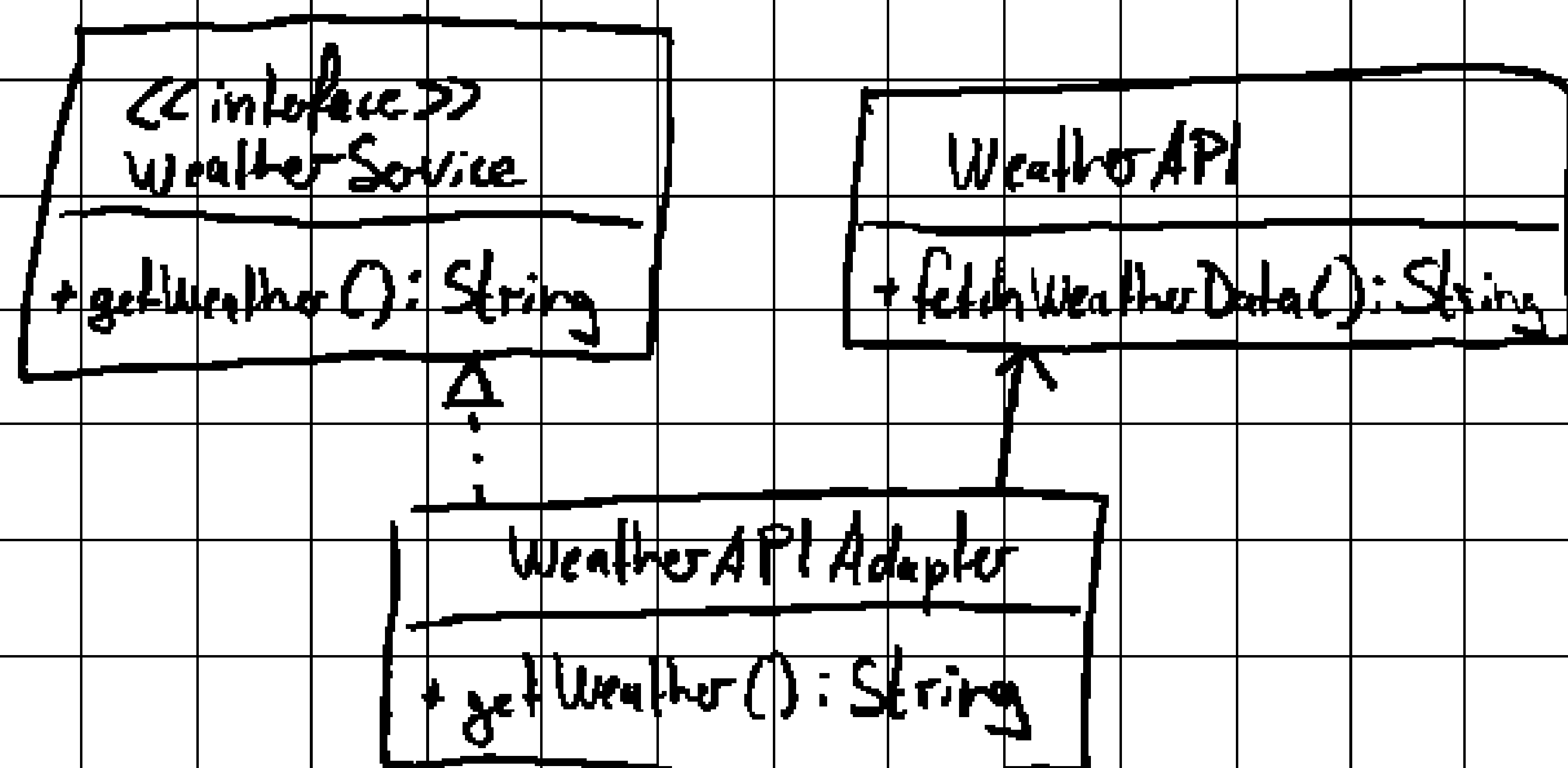
Besteht aus folgenden Komponenten

- Target → Interface, das der User benutzen möchte
- Adapter → Klasse, die Target implementiert und Anfragen übersetzt
- Adaptee → Klasse / Interface, die wir benutzen möchten

Beispiel

Wetter app das eine Api benutzt, welche aber beide andere Methoden haben

UML



Java of WeatherAPIAdapter

```java
public String getWeather() {
    return fetchWeatherData();
}
```

Facade Pattern

↳ einfaches Interface für komplexes System

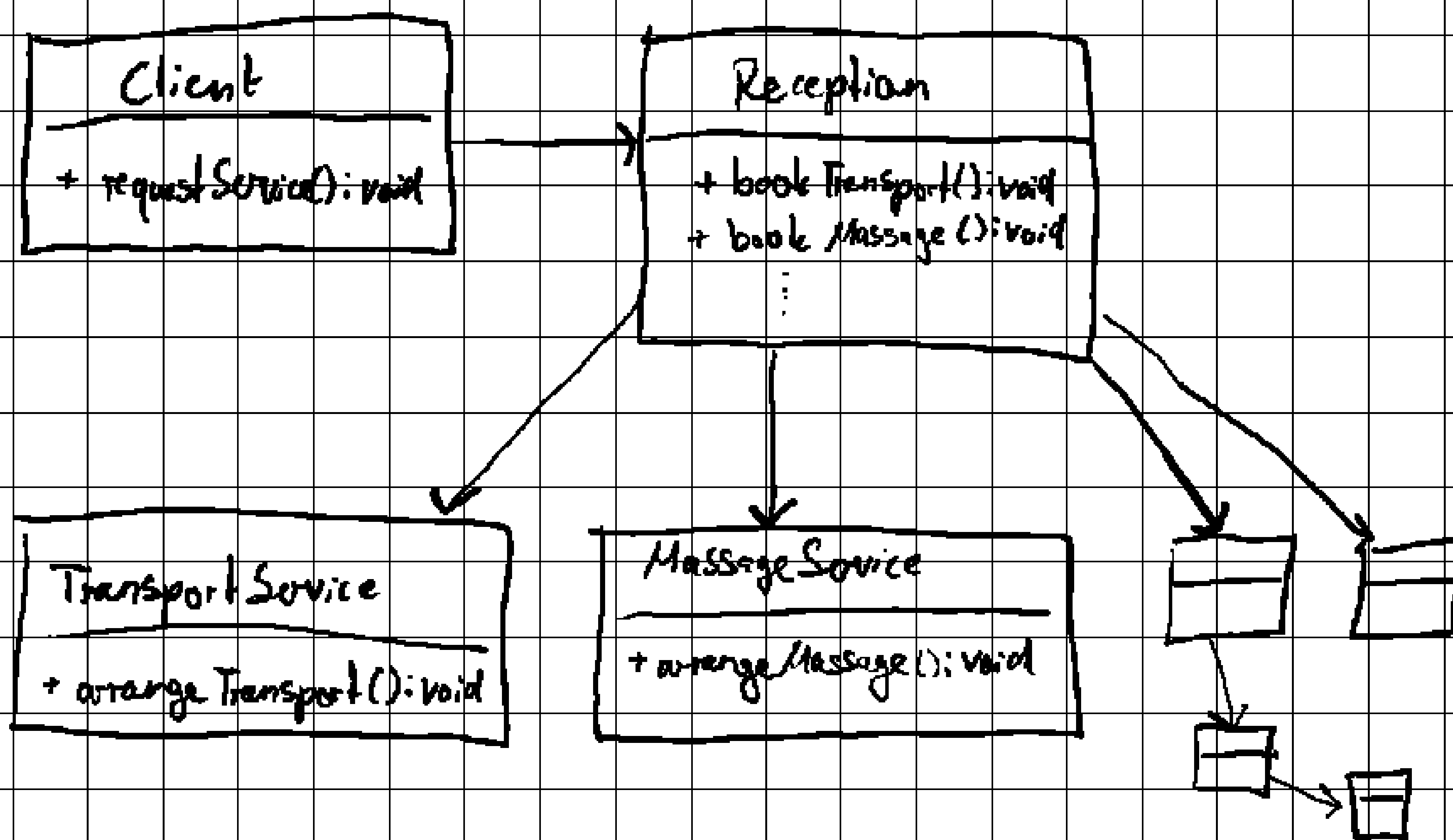
Merkmale

Besteht aus folgenden Komponenten

- viele Subsystem Klassen
- Facade → Interface für access zu Subsystemen

Beispiel

↳ Ein Hotel bietet verschiedene Services an, aber alle können an der Reception gebucht werden



Proxy Pattern

↳ erstellt einen Vertreter für ein anderes Objekt

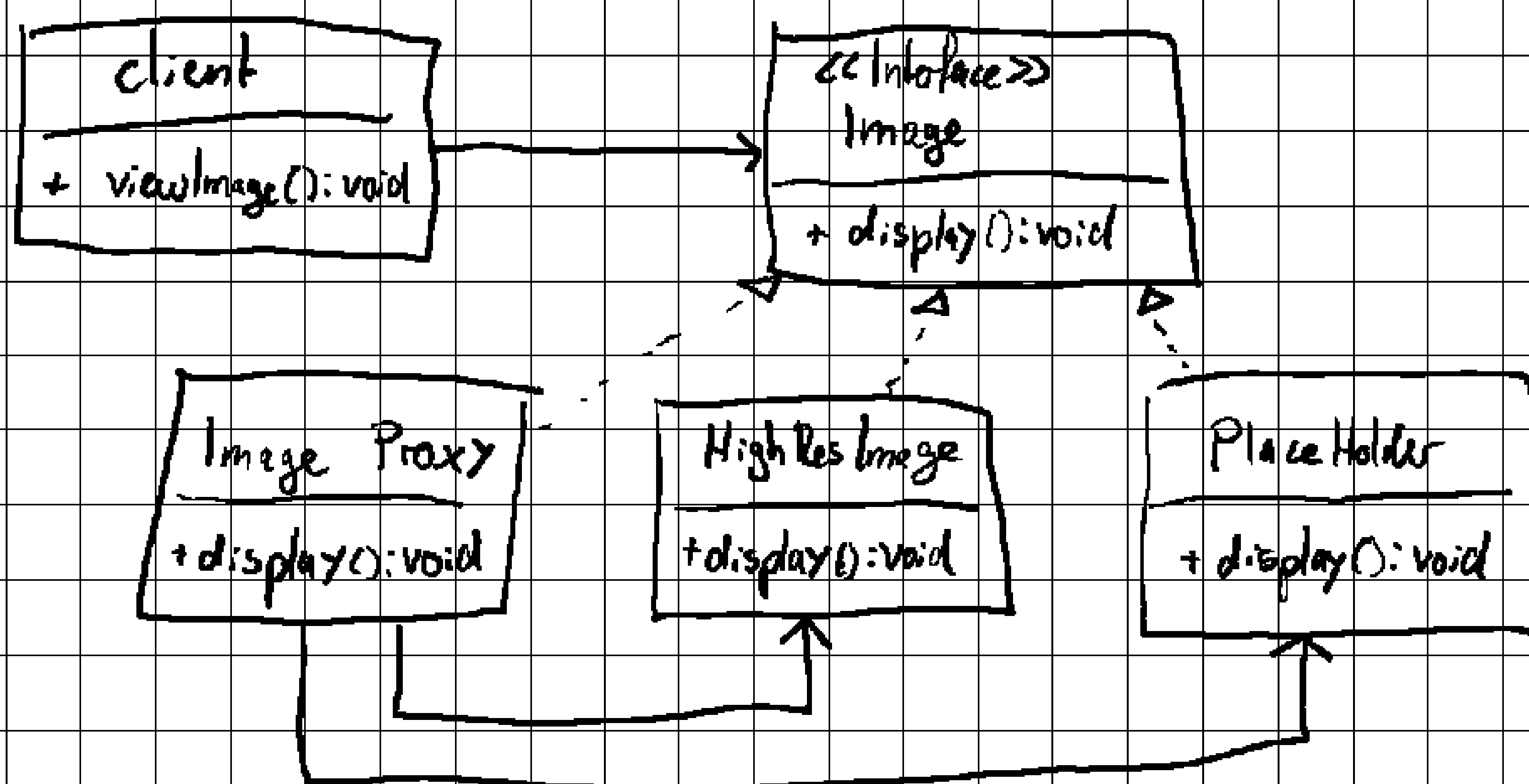
Merkmale

Besteht aus folgenden Komponenten

- Subject → Interface für das eigentliche Objekt
- Proxy → Implementiert Interface und vertritt das Object
- RealSubject → Implementiert Interface

Beispiel

↳ Ein Bild soll angezeigt werden. Jedoch soll ein Platzhalter eingeblendet werden, bis das Bild vollständig geladen hat.



Observer Pattern

↳ Kommunikation einer Status-Änderung an andere Objekte

Merkmale

Besteht aus folgenden Komponenten

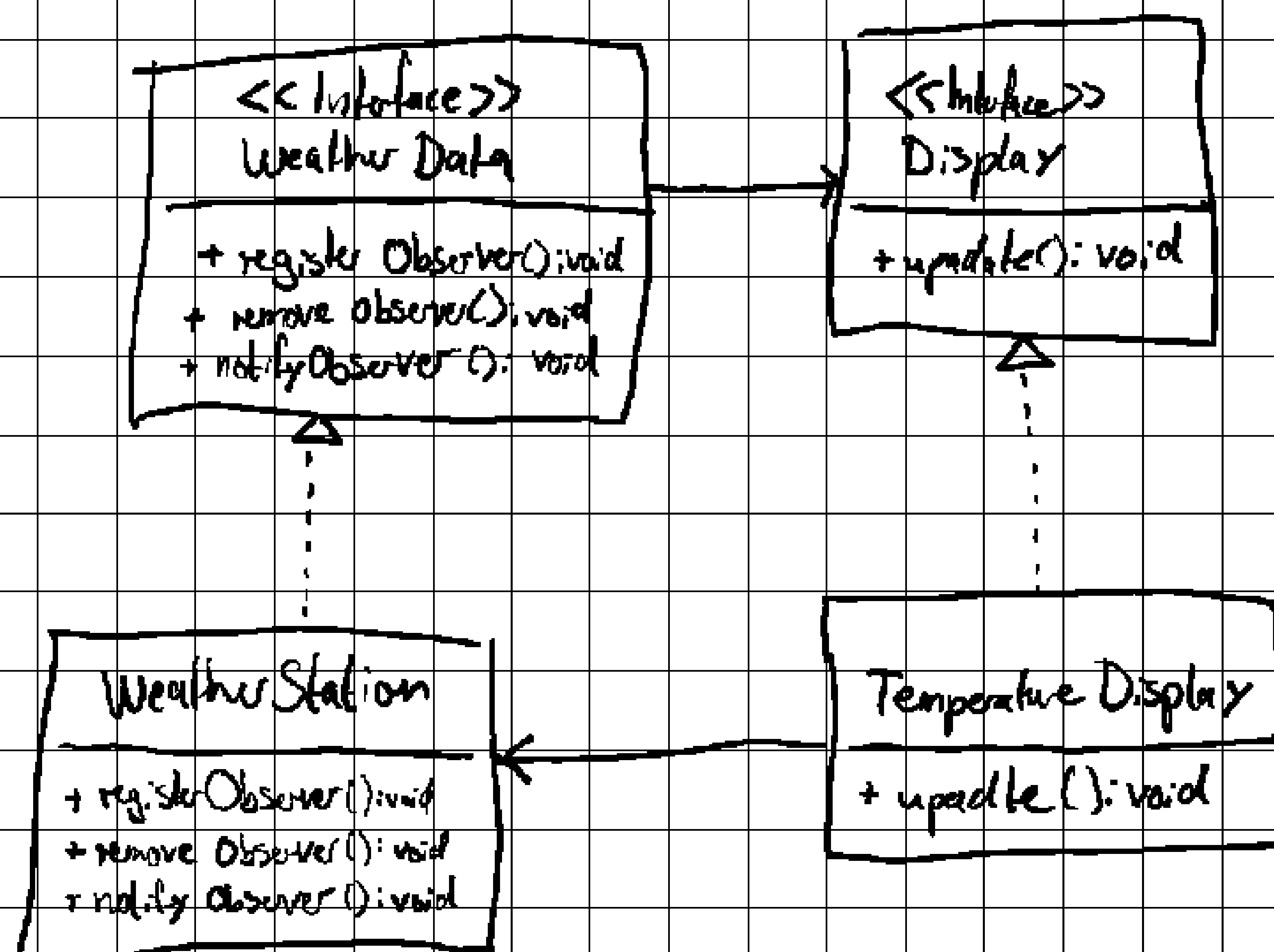
- Subject → Interface für Subjekt
- concrete Subject → Implementiert Subjekt, ist eigentliches Observable
- Observer → Interface für Observer
- concrete Observer → Implementiert Observer, beobachtet Observable

Pull Methode → Update beantragen (Social Media)

Push Methode → Benachrichtigung bei update (subscribed to)

Beispiel

↳ Eine Wetterstation liefert Daten für das Thermometer



Strategy Pattern

↳ Verhaltensänderung eines Objektes wärend der Runtime

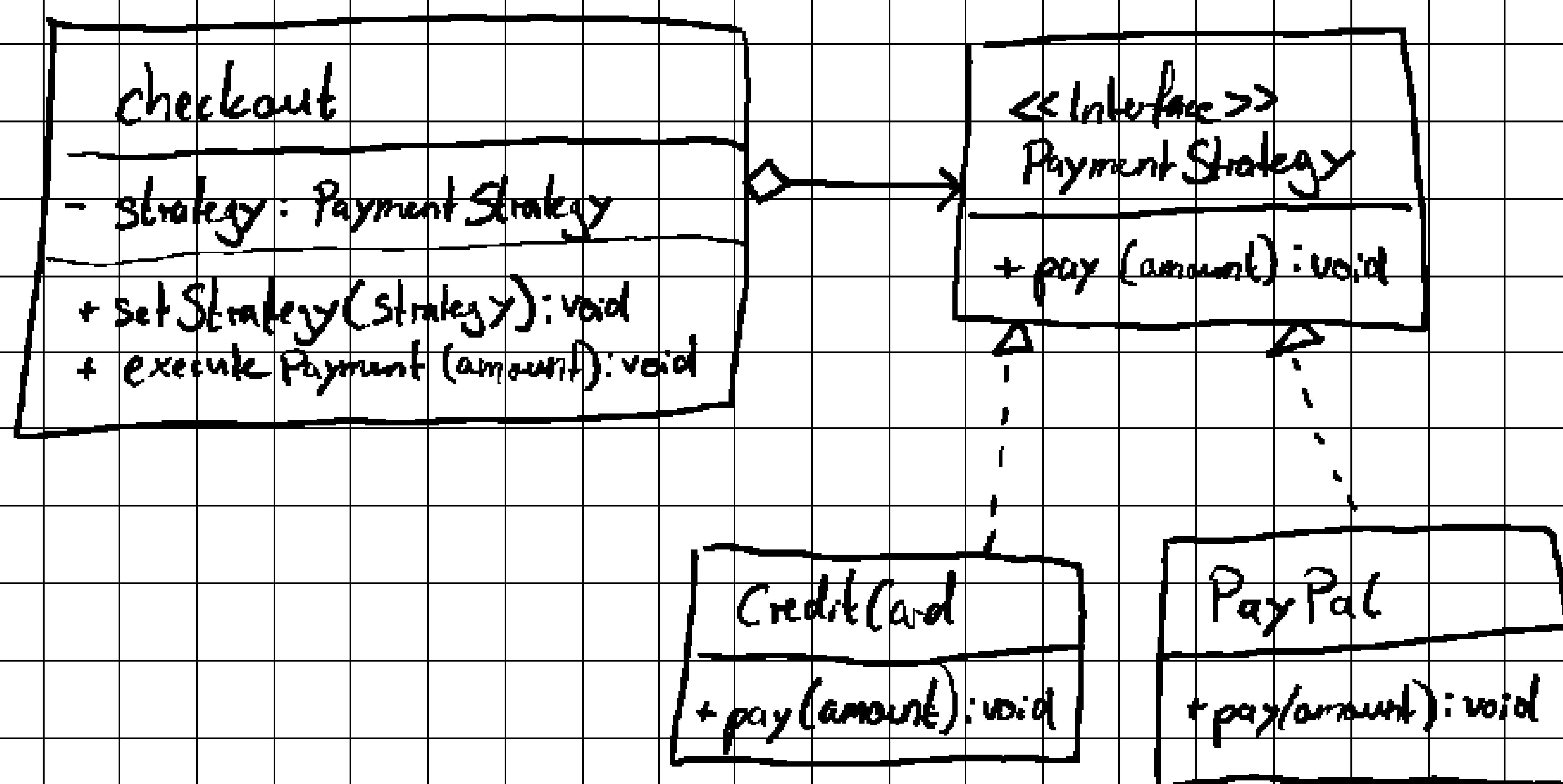
Merkmale

Besteht aus folgenden Komponenten

- Context → Aktion, die einen Algorithmus benötigt
- Strategy → Interface für verschiedene Algorithmen
- concrete Strategy → Implementiert Strategy mit spezifischen Algorithmus

Beispiel

↳ Beim Bezahlen im online Shop kann der user zwischen verschiedenen Zahlungsmethoden auswählen



Template Method Pattern

↳ Skelett eines Algorithmus in einer Methode für mehrere Usecases

Merkmale

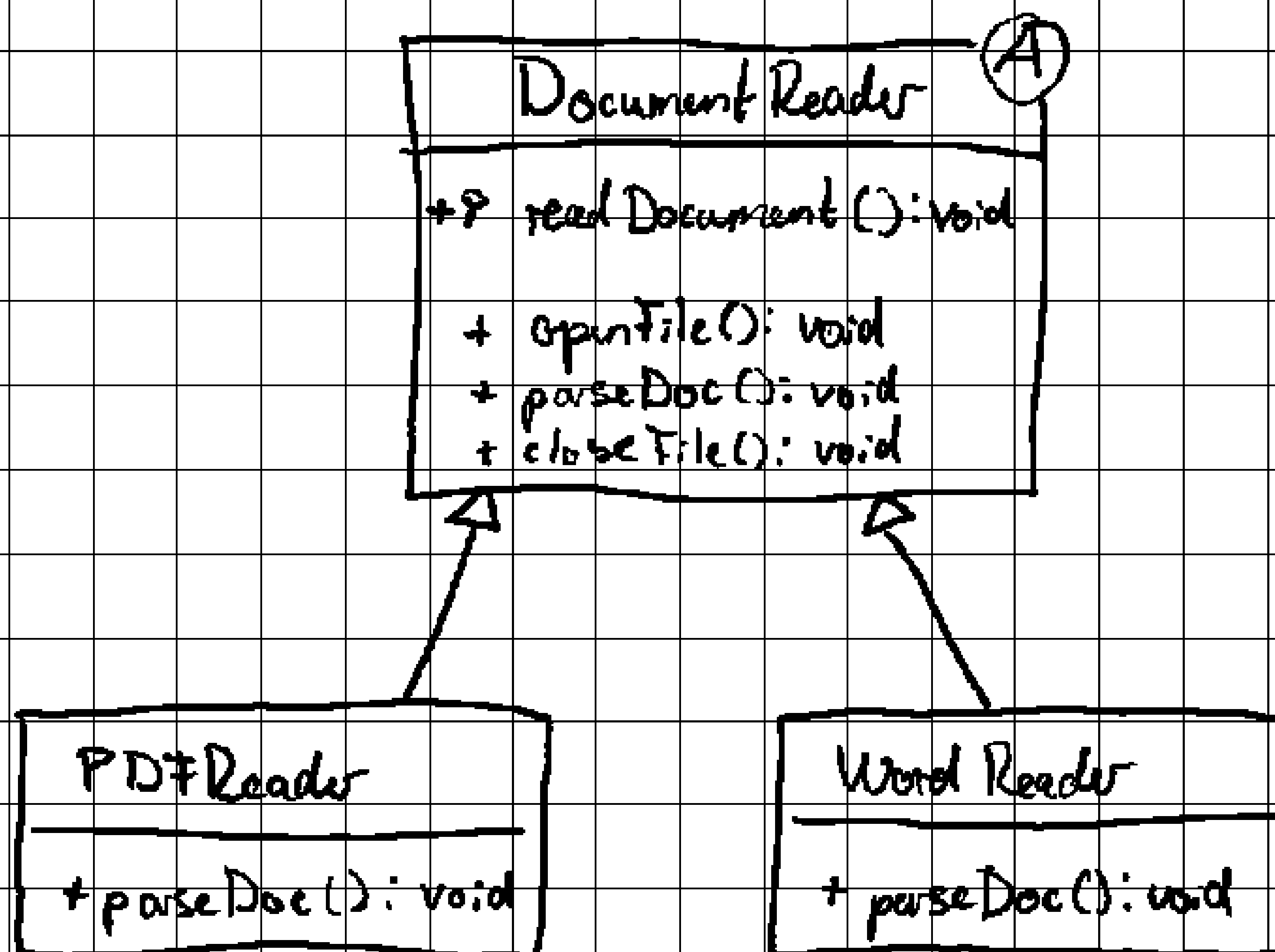
Abstrakte klasse aber ein par Methoden sind final also unveränderbar

Besteht aus folgenden Komponenten

- Abstract Class → Grundgerüst mit variabeln Elementen
- Concrete Class → definiert die variabeln Elemente

Beispiel

Ein Dokumenten-Lese Programm kann mehrere verschiedene Arten von Dokumententypen lesen



Command Pattern

↳ wandelt Anfrage in Objekt um für eine einheitliche Übermittlung

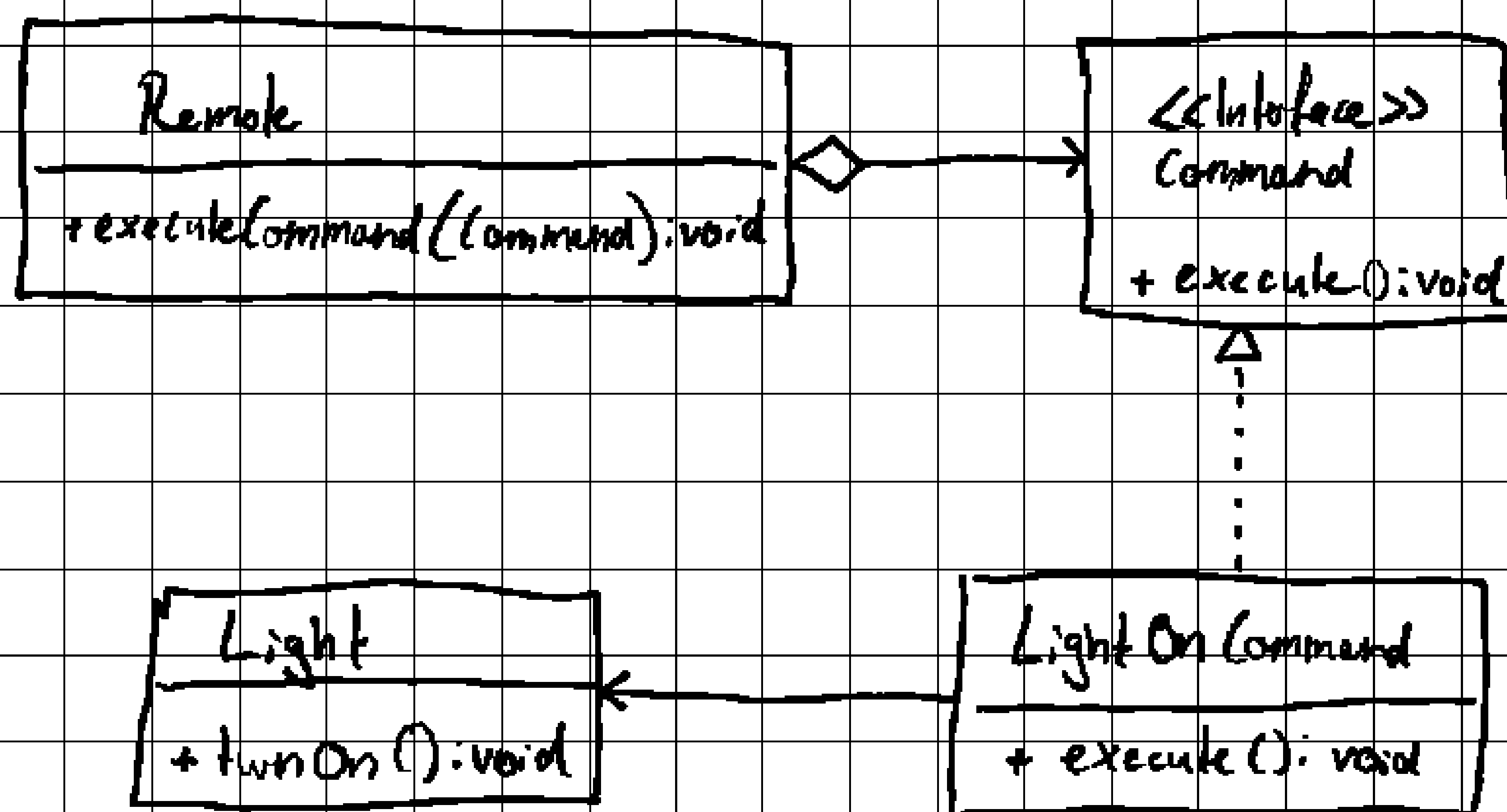
Merkmale

Besteht aus folgenden komponenten

- Command → Interface definiert operationen
- Concrete Command → Implementiert command und definiert Verbindung zwischen receiver Objekt und Aktion
- Receiver → Objekt, dass die Aktion des Commands ausführt
- Invoker → Enthält den Command bittet um Ausführung

Beispiel

Schalte eine Lampe mittels Fernbedienung an/aus



State Pattern

↳ ändert Verhalten eines Objektes basierend auf dessen State

Merkmale

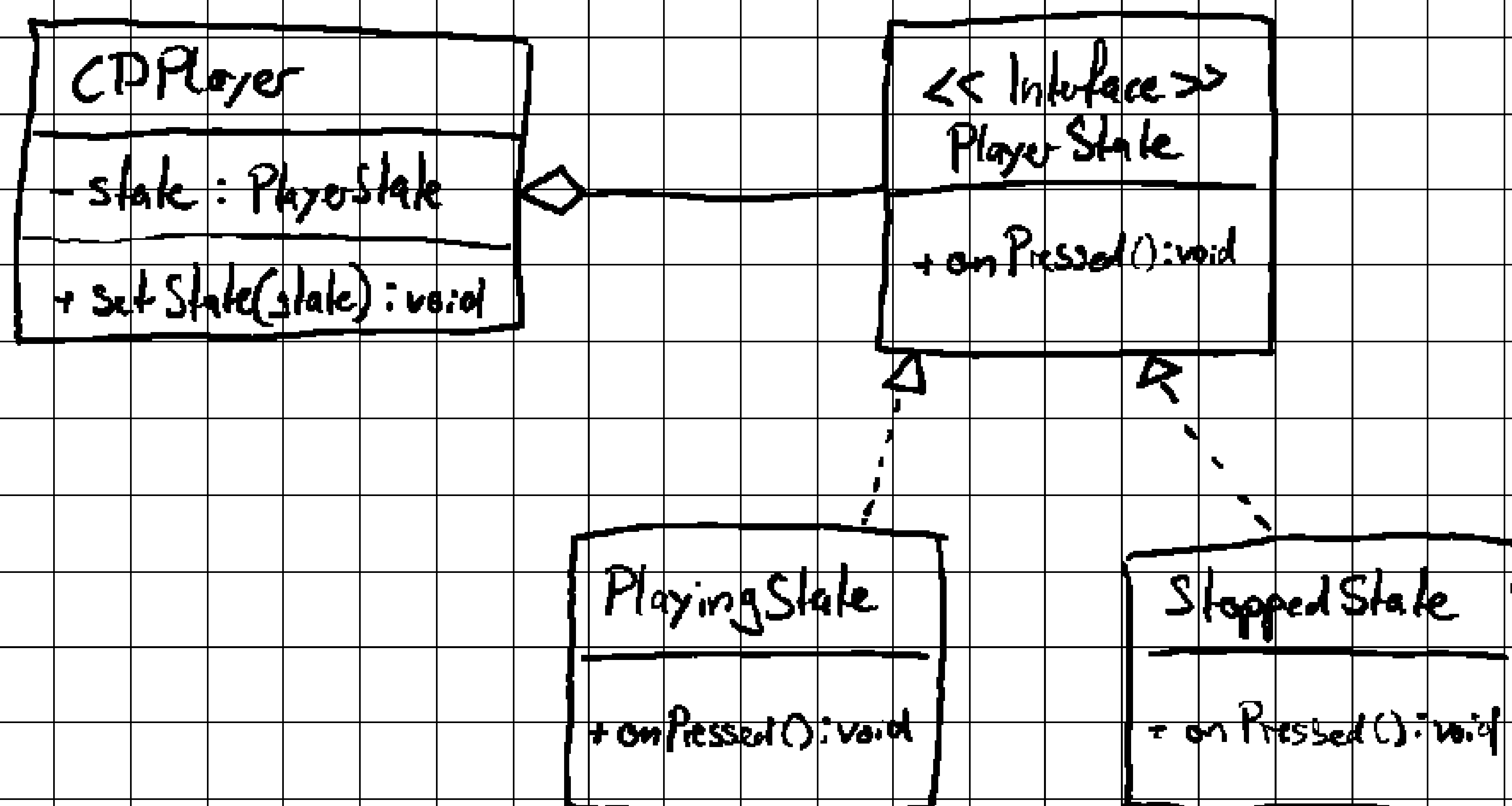
Besteht aus folgenden Komponenten

- Context → Klasse mit State-basierenden Verhalten
- State → Interface definiert Struktur
- Concrete State → implementiert State und definiert Verhalten

Beispiel

CD Player hat einen Knopf für Play/Stop.

Je nach Status des Players wird Stop/Play ausgeführt beim Knopfdruck.



Iterator Pattern

↳ Möglichkeit Objekte einer kollektion der Reihenfolge nach abzurufen, ohne ihre eigentliche Darstellung zu kennen

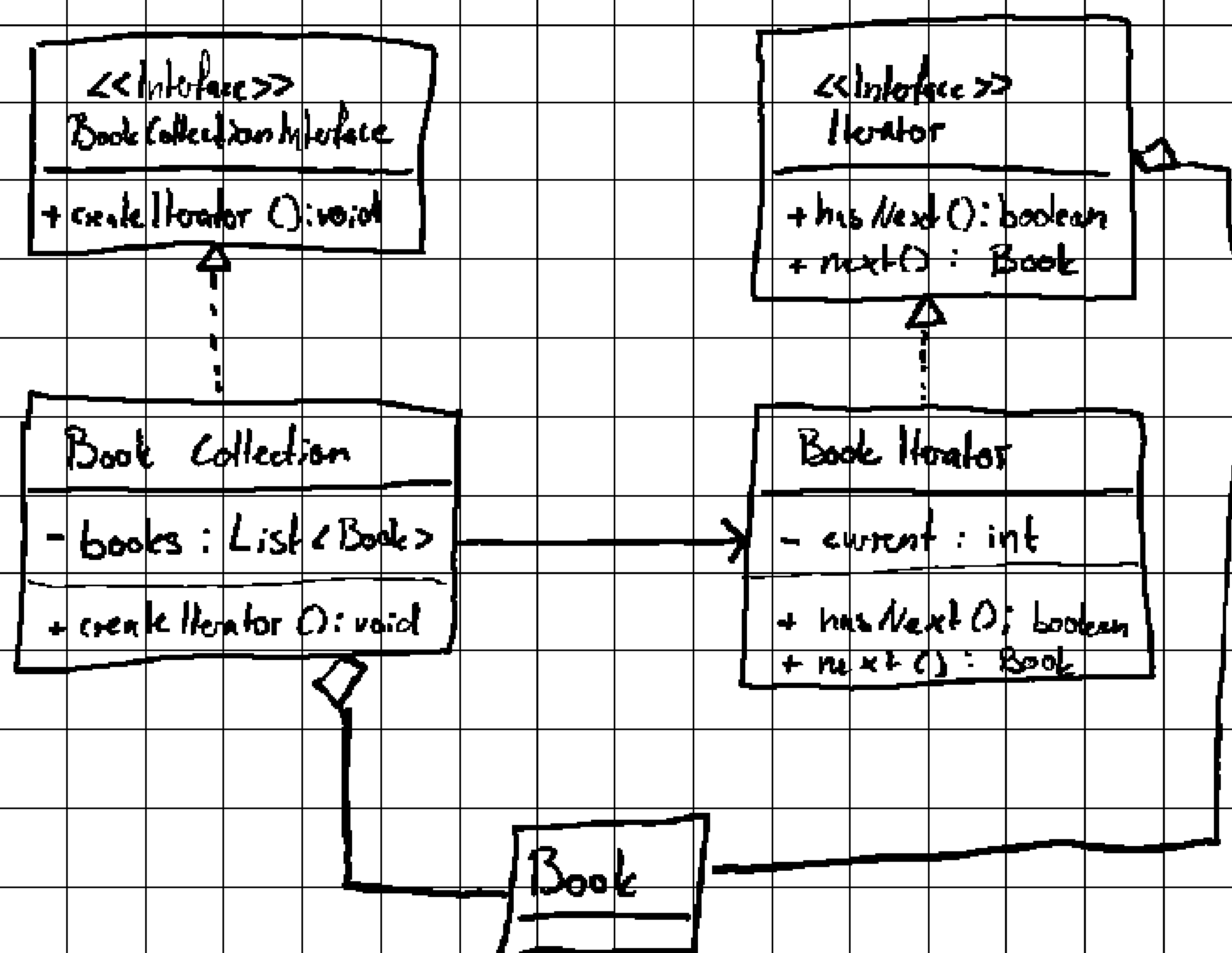
Merkmale

Besteht aus folgenden Komponenten

- Iterator → Interface für allgemeine Iterate Operationen (z.B. hasNext())
- concrete Iterator → implementiert Iterator für spezifische kollektion
- Aggregate → Interface für Erzeugung einer iterierbaren kollektion
- concrete Aggregate → implementiert Aggregate mit Methoden für spezifische kollektion

Beispiel

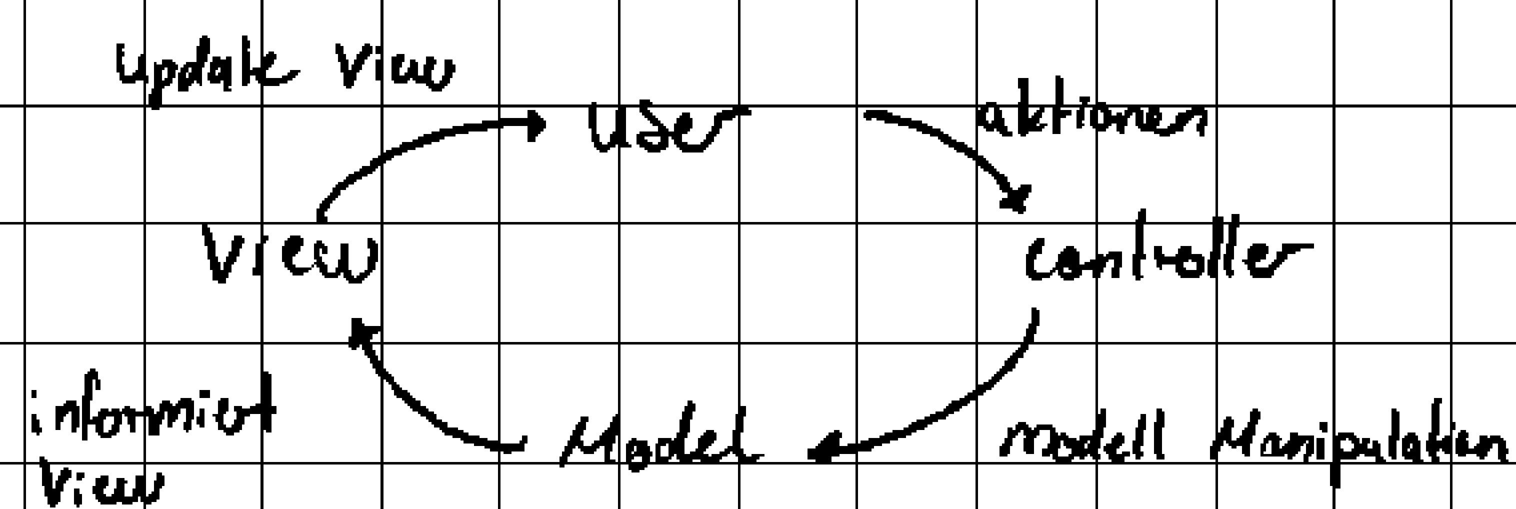
↳ Bücher katalog einer Bibliothek egal wo sich die Bücher zurzeit befinden



Compound Pattern

↳ Kombination aus mehreren Pattern

Model - View - Controller (MVC)



Model: Data Logic

View: Data Presentation

Controller: Data Manipulation

Patterns:

Model → Observer → Objekt update, bei Controller Änderung

View → Composite → Tree Struktur mit Container & Buttons

Controller → Strategy → Interaktion mit Model

Pros & Cons

+ Simultane Entwicklung
+ einfach zu testen
+ reduziert komplexität
+ einfach für Instandhaltung

- Daten abfrage in View ist ineffizient
- Framework navigation wird komplex

Rotting Software (Design Smells)

- Rigidity → Software ist schwer zu ändern
- Fragility → Software geht kaputt bei Änderungen
- Immobility → Software ist nicht modular
- Viscosity → Änderung ist einfacher "hacky" zu implementieren, wie Richtig
- Needless Complexity → unnötige Features fürs grosse Ganze
- Needless Repetition → Copy & Paste code an verschiedenen Stellen
- Opacity → schwer zu verstehen

SOLID Principles

S - Single Responsibility
O - Open / Closed
L - Liskov Substitution
I - Interface Segregation
D - Dependency - Inversion

Single Responsibility Principle

↳ Eine Klasse sollte nur eine Funktionalität haben

  ↳ kein Schweizer-Taschenmesser

Beispiel

↳ Koch in einem Restaurant ist für das kochen verantwortlich, nicht für das Servieren, die Finanzen etc.

Open / Closed Principle

↳ open for extension, closed for modification

  ↳ Features hinzufügen sollte ohne Änderung an Bestehendem gehen

Beispiel

↳ Lego Steine können zusätzlich gekauft und hinzugefügt werden ohne die alten Steine zu ändern.

Liskov Substitution Principle

↳ Objekte einer Superklasse können mit Subklassen-Objekten substituiert werden, ohne das es das Programm beeinflusst.

Beispiel

↳ ersetze eine Glühbirne mit einer Energiespar-Birne. Sie funktioniert, ohne Änderungen an der Lampe selber.

Interface Segregation Principle

↳ kein God-Interface → User sollte nicht auf funktionen angewiesen sein, die er nicht braucht

Beispiel

↳ niemand will eine universal Fernbedienung für alle Geräte mit endlos vielen knöpfen. Lieber mehrere, einfache Fernbedienungen.

Dependency Inversion Principle

↳ High-level & Low-level module sollten nicht auf einander angewiesen sein. Viel mehr sollten sie von einer Abstraktion abhängen

Beispiel

↳ Ein Auto sollte nicht abhängig sein von dem genauen Motor, viel mehr sollte es mit allen Motoren funktionieren, solange diese den selben Aufbau von Protokollen hat.

Code Smells

- Duplicate Code —fixes→ extract Method
- Long Method → extract Method / decompose
- Large class → extract class
- Long Parameter List → replace param with Method / param-object
- Divergent Class → extract class
  ↳ multiple usecases
- Shotgun Surgery → move methods
  ↳ 1 change, x small changes
- Feature Envy → move methods
  ↳ method calls a lot of other methods
- Data Clumps → extract class → make them own object
  ↳ Data in a lot of places
- Primitive Obsession → replace with classes
- Switch Statements → polymorphisms, set up inheritance

Anti - Pattern

↳ schlechte Lösung für Problem

Amelioration - Pattern

↳ weg von schlechter Lösung zu einer Guten

Anti - Patterns

- Golden Hammer → 1 Ansatz für alles
- God Class → 1 Klasse für alles
- Spaghetti Code → keine Struktur
- Gas Factory → unnötig komplex
- Functional Decomposition → kein Objekt-orientiertes Design
- Boat Anchor → unnötiger Code
- Dead Code → nicht genutzte code

Technical Depth

↳ "consequences of your own actions"
  ↳ alles was im nachhinein lange braucht um zu fixen
  ⇒ passiert, wenn man auf schnelle Resultate setzt, statt stabiles Design

Software Qualities

- Quality in End-Use
- Quality in Development / Evolution
- Quality in Operation
- Quality in Business

Software Matrics

- LOC → Lines of Code
- Cyclomatic Complexity → Zyklen in Graph

$$\Rightarrow M = E - N + 2P$$

M: complexity, E: Edges, N: Nodes, P: Connected Components

- Halstead Metric → effort of maintaining a software
- Software Maintainability Index → Instandhaltungs-Schwirigkeit
- Defect Density → Anzahl Bugs per 100 Code Zeilen
  ↳ <0.5 – stable; 0.5-3 – Maturing; 3-6 – Vulnerable; 6-10 – Error-Prone; >10 – Useless

Software Testing

↳ Qualitäts prüfung

Test Dummys

↳ komponenten, welche für Tests benutzt werden

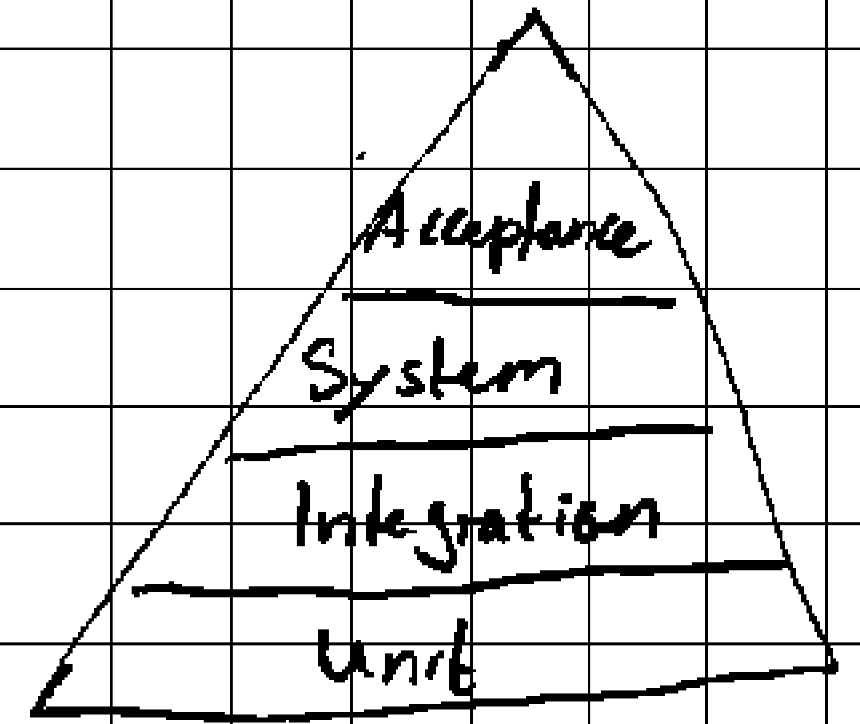
- Mock → fake klasse, die calls registriert
- Stub → fake klasse mit vordefinierten Daten

Testing Methods

- White Box → Tests basieren auf internem Wissen des Codes
- Black Box → Tests basieren auf externem Wissen
- Grey Box → Test von Modulen & deren Zusammenhang

Testing Pyramid

1. Funktionalität
2. Integration der Module
3. System
4. user needs



Write Tests

- Equivalenz - klassen
  ↳ alle möglichen Input daten in Gruppen
  (Bsp. unter limit, zwischen limits, über limit)
- Grenzwerte
  ↳ genaue Grenzwerte der Inputbeschränkung testen

Monkey Testing

- Dump Monkey → testet wilde inputs (0 Ahnung vom Programm)
- Smart Monkey → testet spezifische Inputs (kennt Programm)
- Brilliant Monkey → testet edge cases (user perspective)

Test Coverage

↳ Abdeckung der tests messen

Verhältnis getested / alle Möglichkeiten

- Funktionen
- Statements
- Äste
- Wege
- Zustände

FIRST Rule of clean Tests

F - Fast → schnelle Ausführung

I - Independent → Tests sollten nich voneinander abhängen

R - Repeatable → Im gleichen Umfeld wiederholbar

S - Self-Validating → True / False output

T - Timely → schnell zu schreiben

JUNIT Cheat Sheet

key words	Assertions
@ Test	assertEquals (expected, actual);
@ Before All	assertSame (exp, act);
@ Befor Each	assertTrue (actual);
@ After All	assertNotNull (actual);
@ After Each	assertArrayEquals (exp, act);

⇒ mehr Möglichkeiten mit Hamcrest Library

Persistency

↳ kann erreicht werden durch Anbindung an eine Datenbank mittels diversen APIs, wie z.B. JDBC oder Datenbankframeworks wie z.B Hibernate

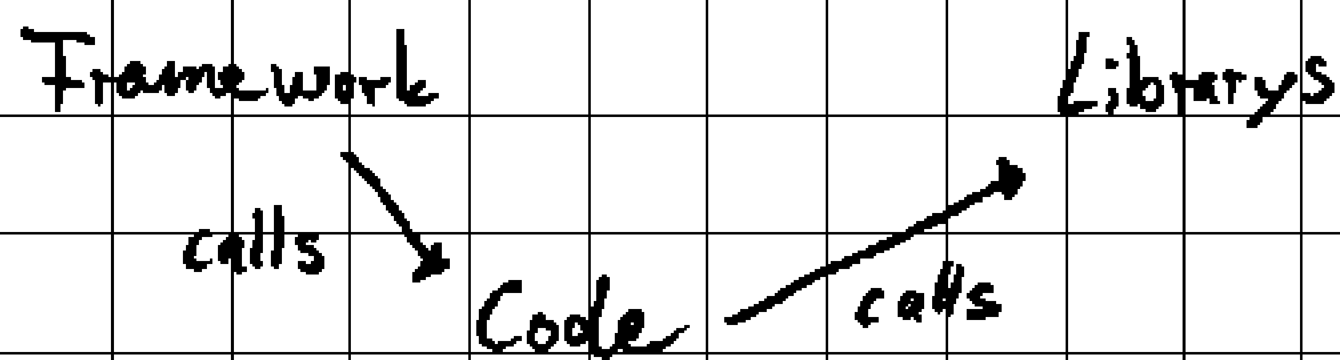
↳ ORM = Object Relational Mapping

Framework

↳ Vorgefertigte Sammlung von Code, die eine Struktur und Richtlinien beinhaltet, um die Entwicklung einer Applikation zu vereinfachen.

Inversion of Code (IoC)

↳ Hollywood Principle → Don't call us, we call you



Template Method

↳ Frameworks haben Templates für ihre Objekte

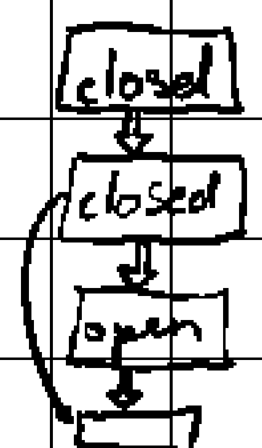
z.B Button → user gibt nur noch function call an

Software Architecture

- Software component → Funktionalitäts Subset
- Software connector → beeinflusst & reguliert interaktionen von komponenten
- architectural configuration → spezifische Assoziationen zwischen component & connector
- architectural style → Kollektion von design choices
- architectural pattern → design choices zur Lösung von Problemen

Architectural Pattern

- Model - View - Controller
- Pipes & Filters → Kollektion von Filtern zur Datenverarbeitung
- Layered Architecture → Aufteilung in Subtaskgruppen
  ↳ kommunikation nur mit Nachbarn
  z.B Network Stack



- Client - Server → Server hat service, den client requesten kann
- Monolith → Single Codebase mit mehreren Layern
- Modulith → modularer Monolith
- Microservices → basiert auf Service komponenten mit jeweils eigener Datenbank
- Event Driven → asynchrone event verarbeitung
  → publish / subscribe durch Event Broker

Anti Pattern

z.B Frozen Caveman → angst vor Neuem wegen alter schlechter Entscheidung

Future Proof Software Systems

- dependable
- general Business value
- maintain high changebility (für neue Features)

Software Evolution Process

New Requirements ⇨ Specification ⇨ Development ⇨ Integration

↳ Qualität resultiert aus Architektur-, Design,- Implementation- Choices

Devils of System Engineering

- complexity → wärend Entwicklung beachten
- change → Änderungen müssen organisiert & koordiniert passiven
- uncertainty → müssen besprochen und beobachtet werden

⇨ können nur gemanaged werden, nicht bekämpft

Resilience

- absorb → recover → sustain

⇨ muss geplant in die Software integriert werden

Safety

↳ Anhaltender Zustand, der Gefahren Risiko unterhalb eines akzeptablen Levels hält.

Software Evolution

↳ ohne Strategie nimmt die Qualität mit der Zeit ab.

↳ Architectural Erosion (z.B COBOL im Banking)

Managed Evolution

- Business value, changeability & dependability werden kontinuierlich weiter entwickelt
- Sie werden ebenfalls mit verschiedenen Kennzahlen getracked
- Alle Qualitätsmerkmale sind notwendig
- Die Entwicklungsschritte werden Risiko geprüft

⇨ **Konflikt:**

time to market & Development Cost vs. Clean Implementation