

DESIGN PATTERNS

ZooRoyal IT

Sebastian Knott

10/26/2018

WAS SIND PATTERN?

GRUNDIDEE VON PATTERN

- ❖ Alle Ingenieursdisziplinen benötigen eine **Terminologie** ihrer wesentlichen Konzepte ...
- ❖ ... sowie eine **Sprache**, die diese Konzepte in Beziehung setzt
- ❖ **Design Pattern** (Entwurfsmuster) wurden zuerst im Bereich der Architektur beschrieben

ZIEL VON SOFTWARE PATTERN

- ❖ Dokumentation von Lösungen wiederkehrender Probleme, um Programmierer bei der Softwareentwicklung zu unterstützen
- ❖ Schaffung einer gemeinsamen Sprache, um über Probleme und ihre Lösungen zu sprechen
- ❖ Bereitstellung eines standardisierten Katalogisierungsschemas um erfolgreiche Lösungen aufzuzeichnen

ABER ...

- ❖ Bei Pattern handelt es sich nicht um eine Technologie (wie z.B. bei UML)
 - ❖ Patterns sollen unterstützen, nicht einengen
- ❖ Es ist eine Kultur der Dokumentation und Unterstützung guter Softwarearchitektur
- ❖ Pattern sind nicht die Lösung für alles

FACADE-PATTERN



PROBLEMSTELLUNG

Der direkte Zugriff auf die Interfaces von ein oder mehreren Subsystemen ist unpraktikabel oder nicht erwünscht.

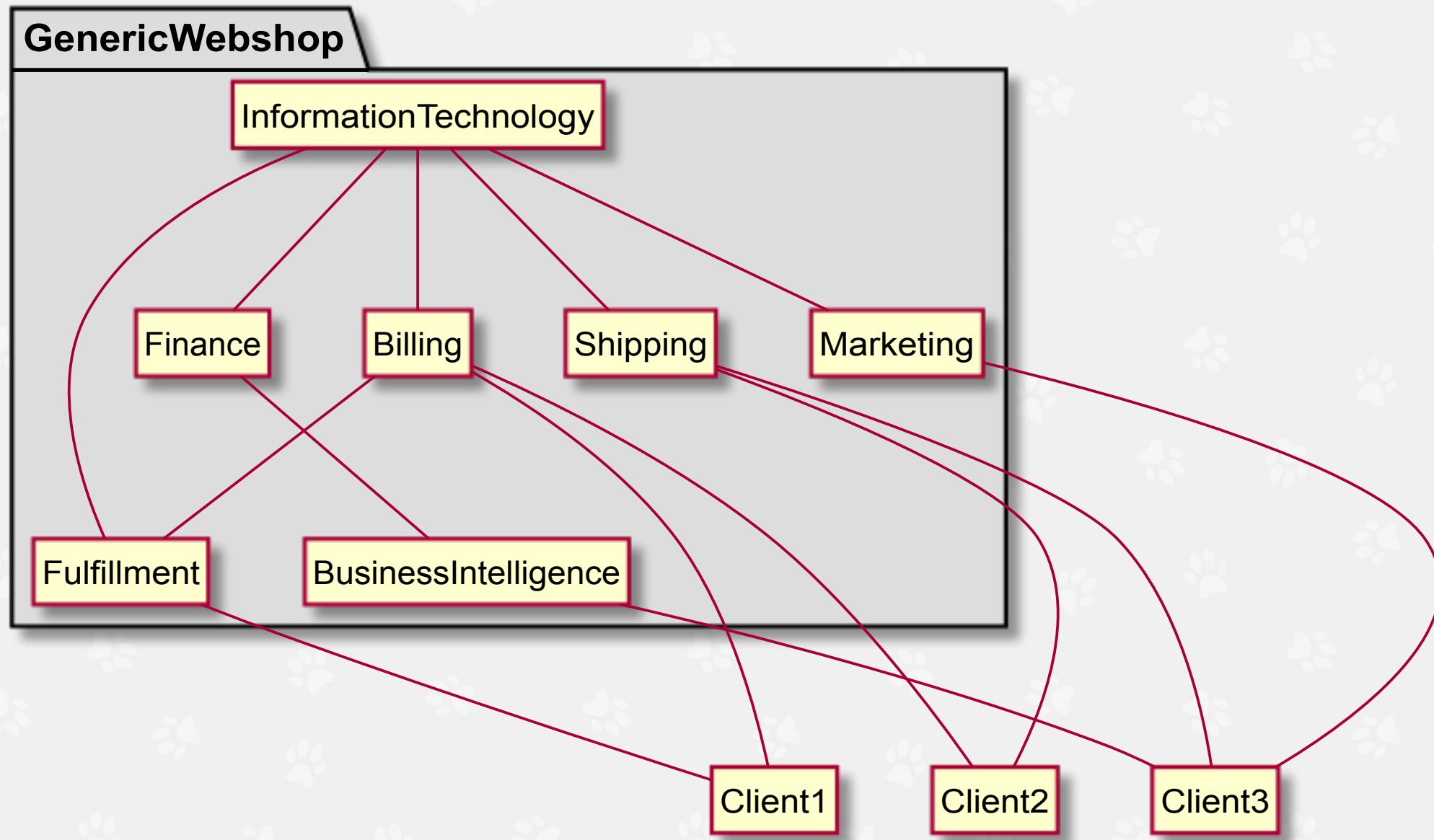
ABSICHT

*Provide a unified interface to a set of interfaces in a subsystem.
Facade defines a higher-level interface that makes the subsystem
easier to use.*

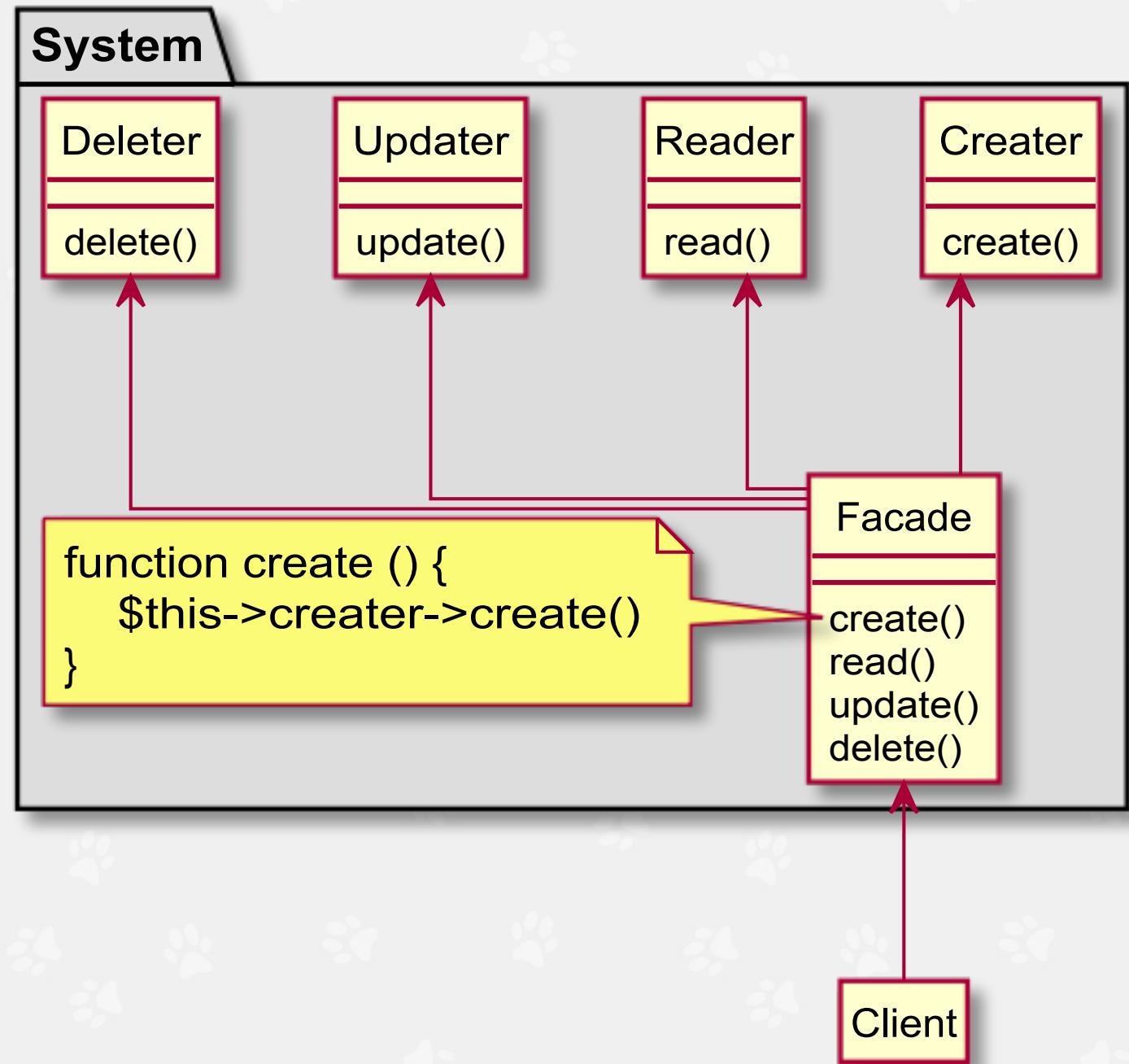
— *Gang of Four* 

- ❖ Interfaces von mehreren Subsystemen bündeln
- ❖ Interfaces in einem besser zugeschnittenen Interface verpacken
- ❖ Trennung von Implementierung und externem Interface

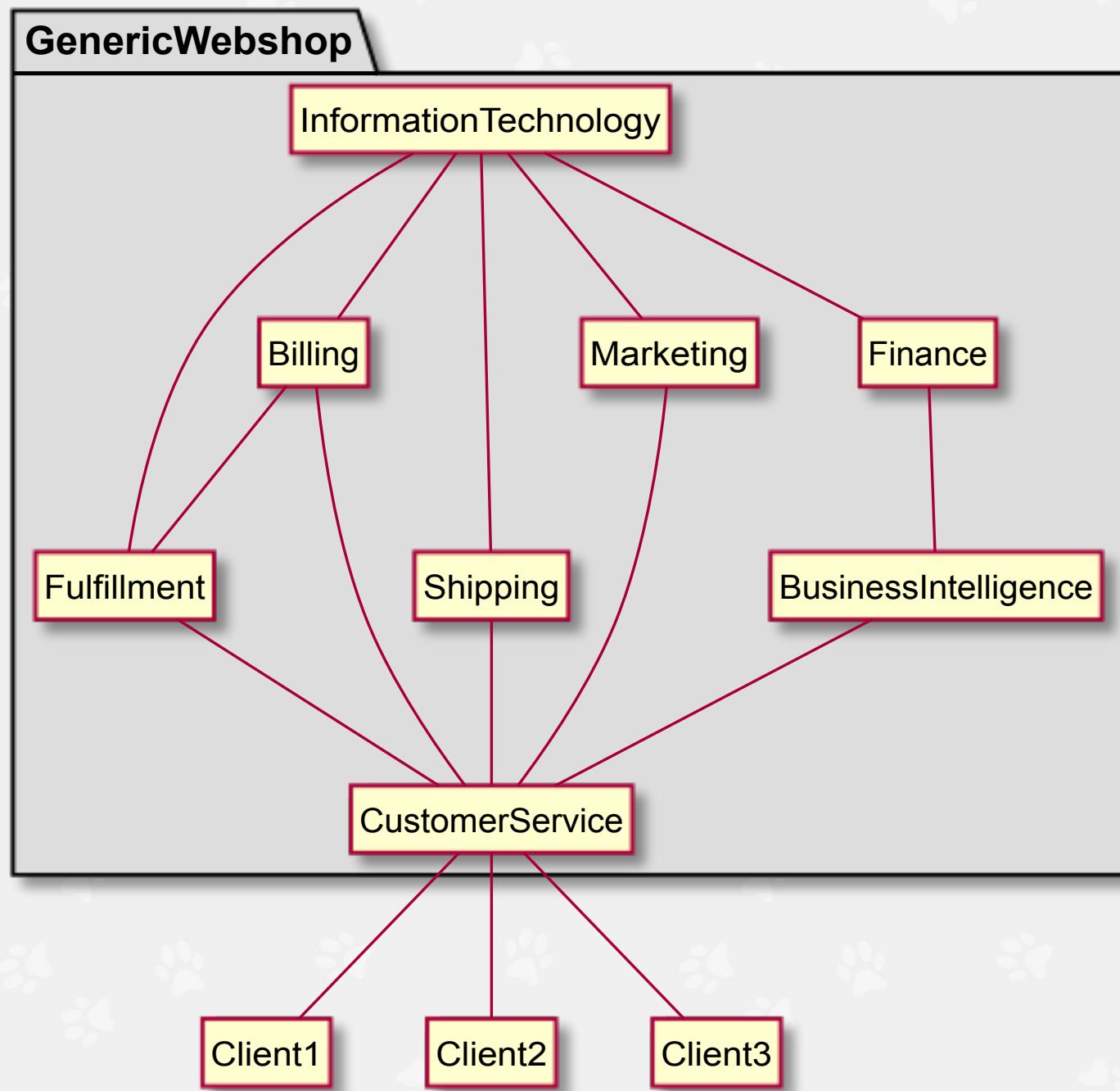
PROBLEMBEISPIEL



STRUKTUR



STRUKTURBEISPIEL



CHECKLISTE ZUR IMPLEMENTIERUNG

1. Planung eines simpleren Interfaces für ein Subsystem
2. Schreiben einer **Wrapper-Klasse** um das Subsystem
3. Interaktionen mit den Subsystemen im Wrapper zusammenfassen
4. Wrapper delegiert Aufrufe an Methoden des Subsystems
5. Clients auf Wrapper umstellen



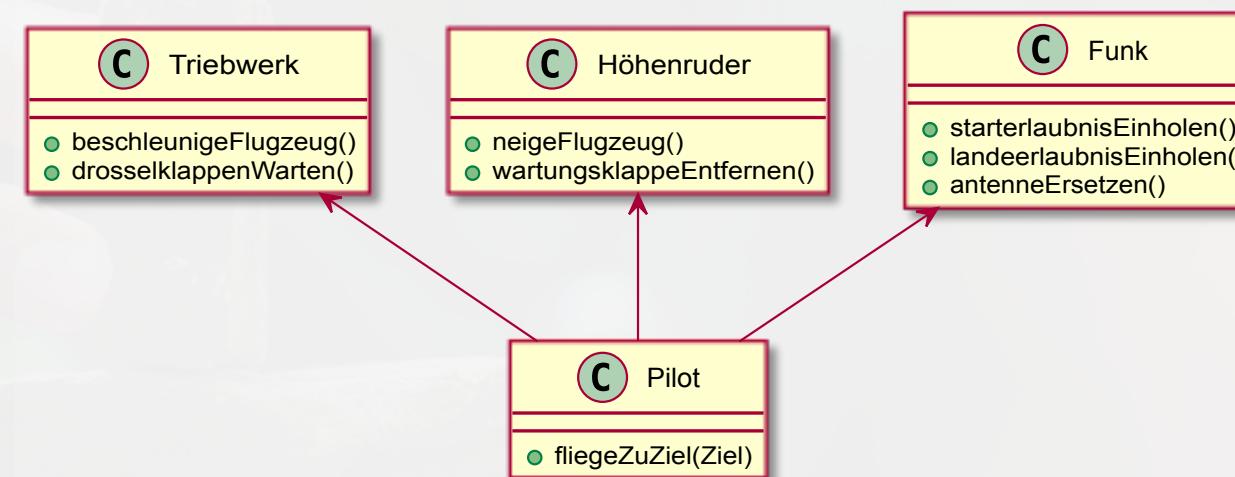
FOLGEN

- Reduziert die Anzahl der zu handhabenden Objekte
- Fördert lose Kopplung
- Verhindert nicht, dass Clients Subsystemklassen verwenden



M	T	W	T	F	S	S
1						
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31					

HANDS-ON



- ❖ Vereinfache das System für den Piloten
- ❖ Benenne die Komponenten der Lösung

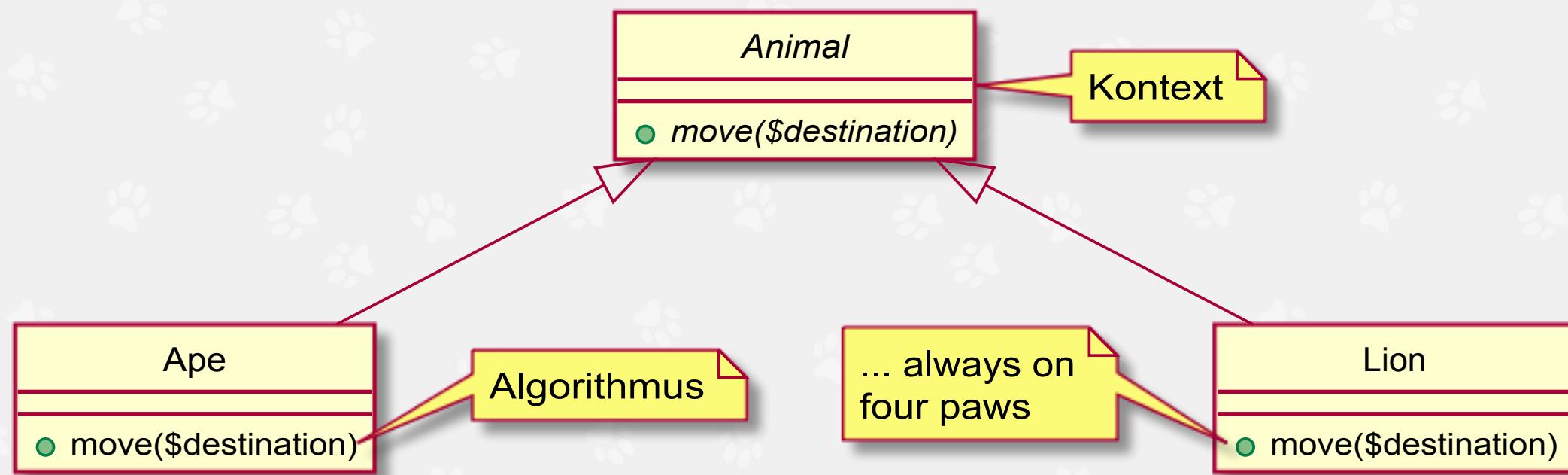
STRATEGY-PATTERN



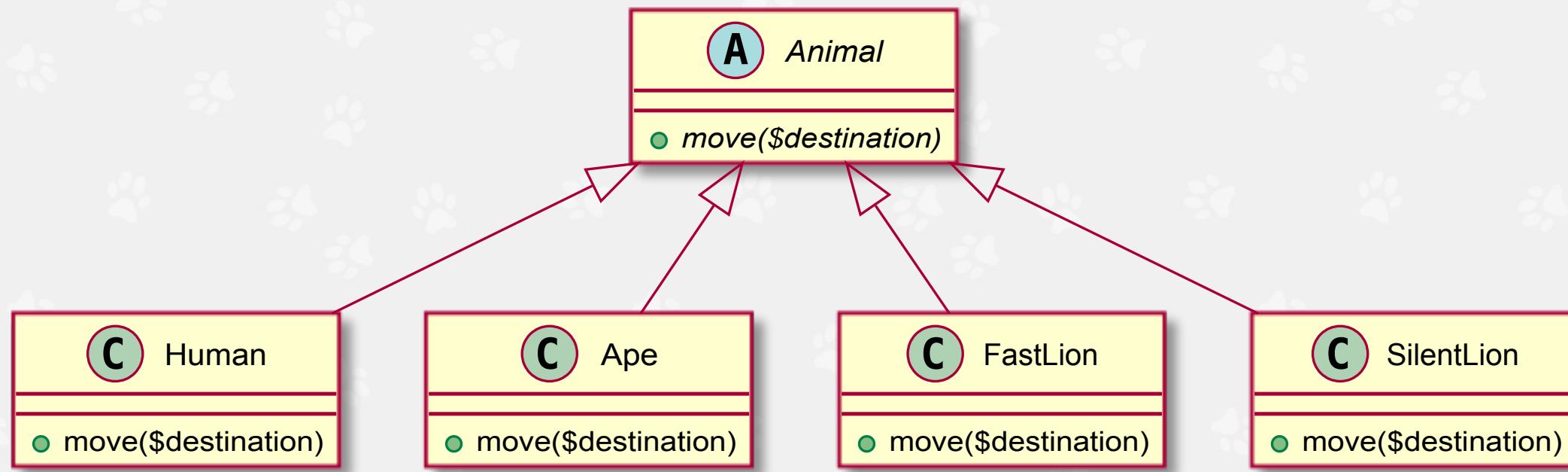
PROBLEMSTELLUNG

Kontext und Implementierung eines Algorithmus sind so fest gekoppelt, dass der Code schwer zu erweitern ist.

- Kontext und Logik sind miteinander verzahnt
- Es ist sehr schwierig Varianten zu bilden
- Verschiedene Logik ist nicht klar voneinander getrennt
- Naives Vererben hilft einem nicht weiter



- ❖ Abstrakte Methode `move()` wird in Kindern implementiert
- ❖ Implementierung unterschiedlich, also nicht in abstrakter Klasse



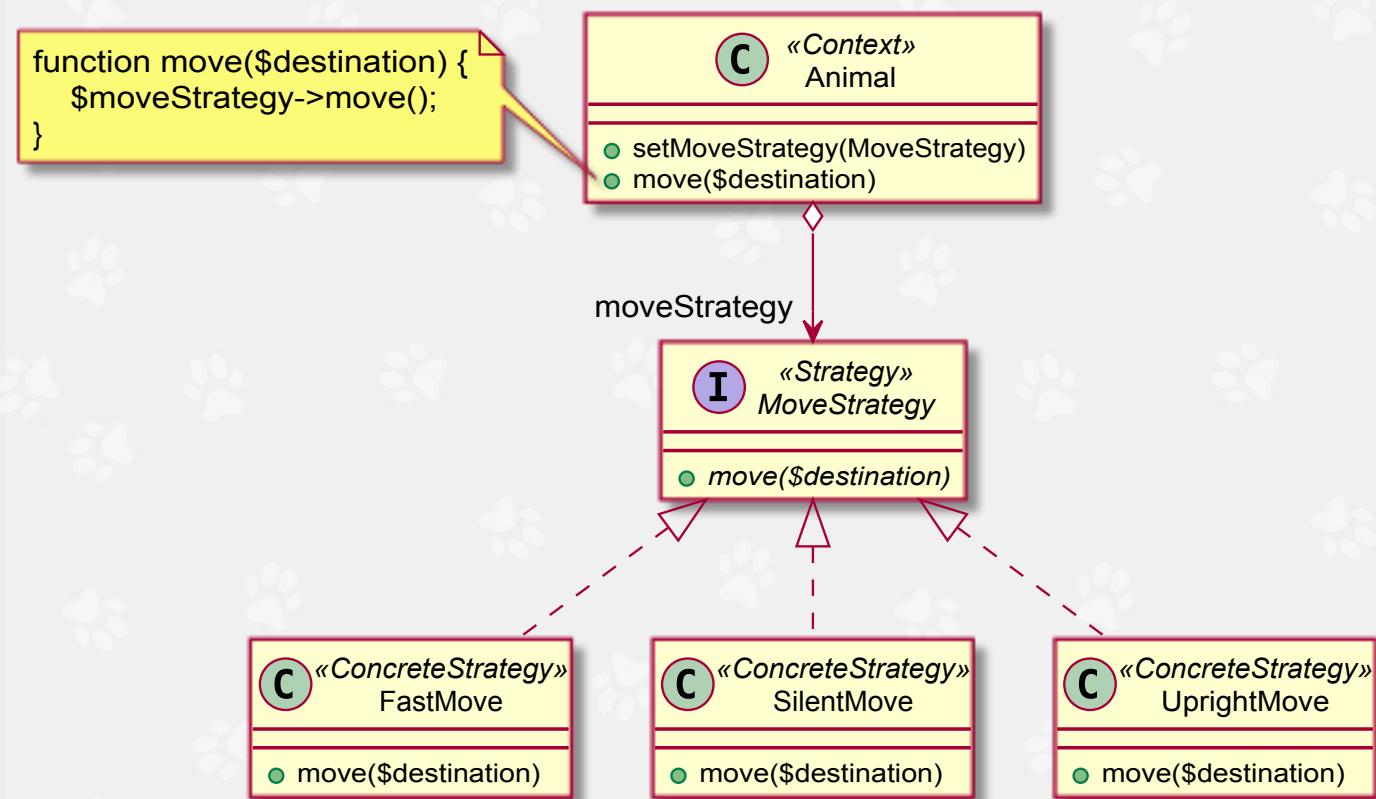
- ❖ Mit der Zeit kommen mehr Implementierungen dazu
- ❖ Jede Klasse braucht eigene Implementierung

ABSICHT

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

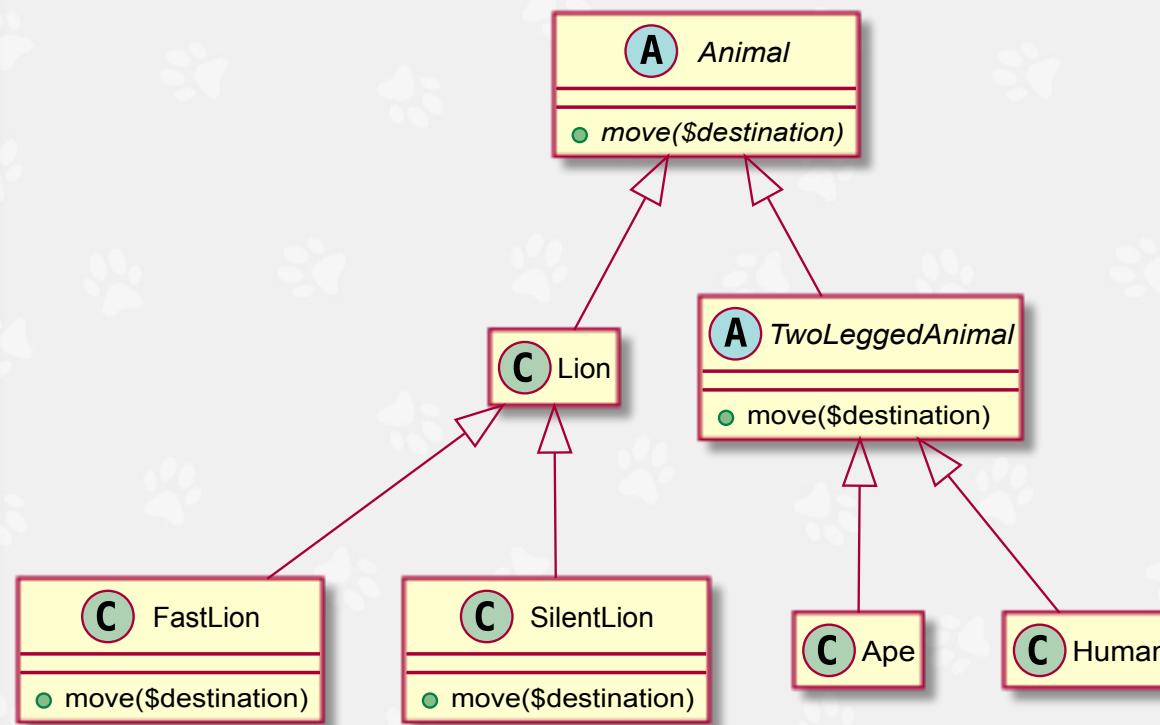
— Gang of Four 

STRUKTUR



- Implementierung gegen Interfaces statt konkrete Klassen
- Kontext von Algorithmus trennen
- Algorithmus austauschbar
- Vermeidet Minimalunterschiede in Klassenvererbung
- Kann zur Laufzeit konfiguriert werden

WARUM NICHT SO... ?



Hieraus ergeben sich weitere Probleme, die sich auf das [Bridge-Pattern](#) beziehen.

CHECKLISTE ZUR IMPLEMENTIERUNG

1. Identifizieren der Menge von Algorithmen zwischen denen variiert werden soll
2. Spezifizieren der Signaturen der Algorithmen im Interface
3. Implementieren der Algorithmen in entsprechenden Klassen
4. Alle Client-Referenzen umstellen auf das Interface

FOLGEN

PRO

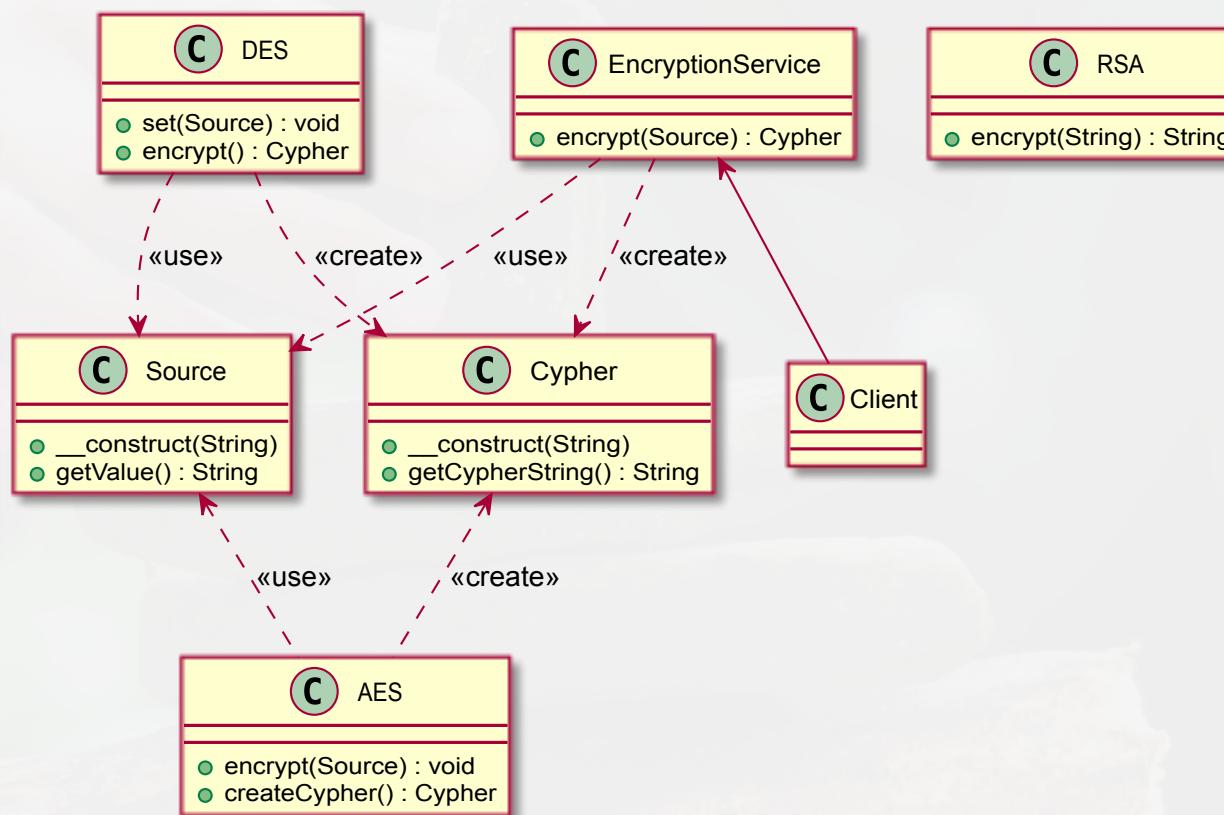
- ❖ Vermeidet Vererbung
- ❖ Kontext und Algorithmus sind jeweils wieder-verwendbar
- ❖ Alternativen zum selben Algorithmus
- ❖ Strategien entfernen Switch-Case aus dem Kontext

CONTRA

- ❖ Erhöht u.U. Klassenanzahl
- ❖ Enge Kopplung zwischen Kontext und Algorithmus
- ❖ Kontext muss in einem Setup-Schritt Strategie übergeben werden



HANDS-ON



Verschlüsselungsdienst kann Quelle verschlüsseln

Verschlüsselungsdienst kann für einen Algorithmus konfiguriert werden

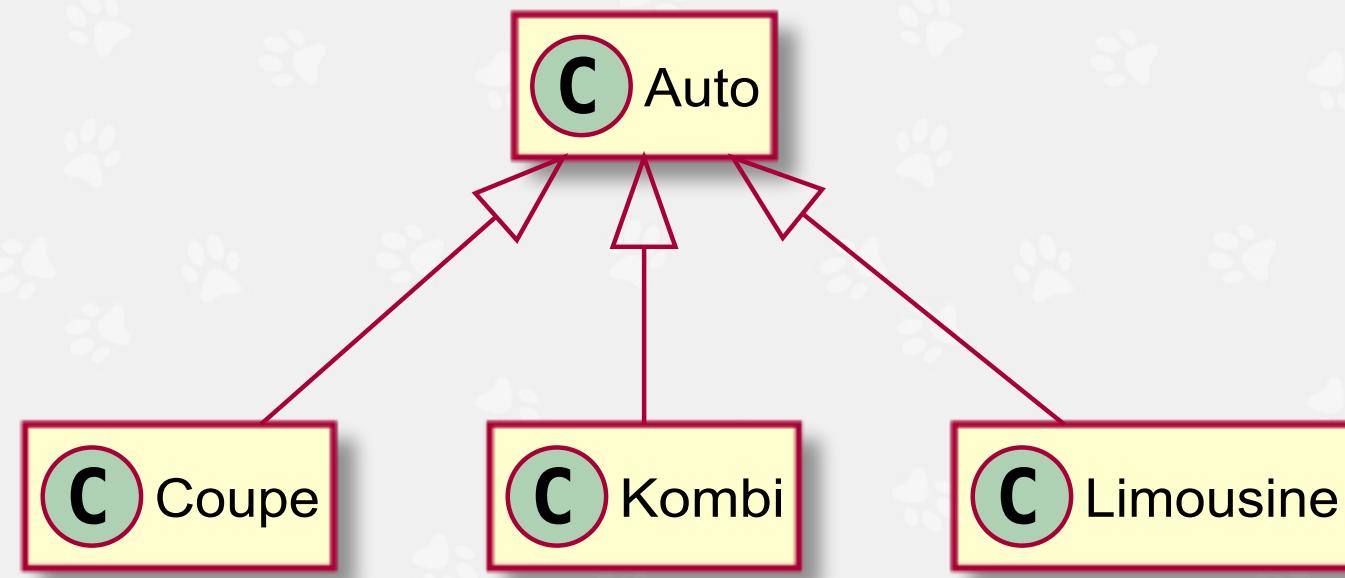
Algorithmen : AES, DES, RSA

BRIDGE-PATTERN



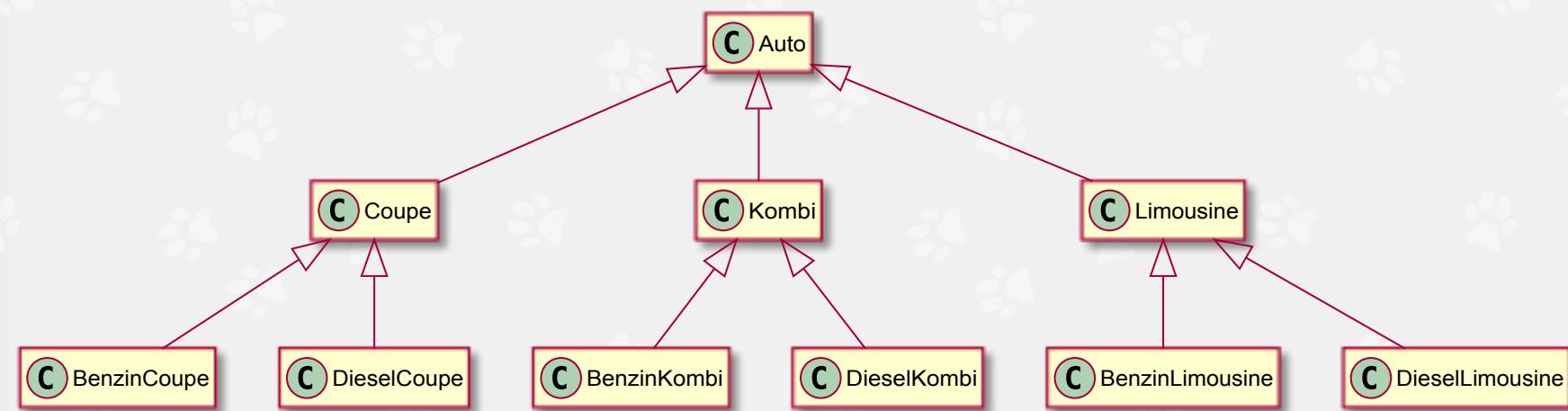
PROBLEMSTELLUNG

Eine Menge von Klassen hat viele orthogonale (unabhängige) Eigenschaften. Eine Vererbungsstruktur würde zu einem unpraktikabel großen Klassenbaum führen.



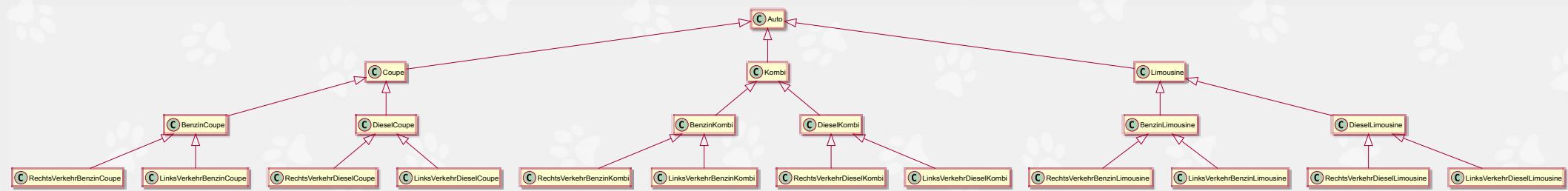
Orthogonale Eigenschaften:

• Baufom



Orthogonale Eigenschaften:

- ❖ Baufom
- ❖ Motorart



Orthogonale Eigenschaften:

- ❖ Bauform
- ❖ Motorart
- ❖ Links-/Rechtsverkehr

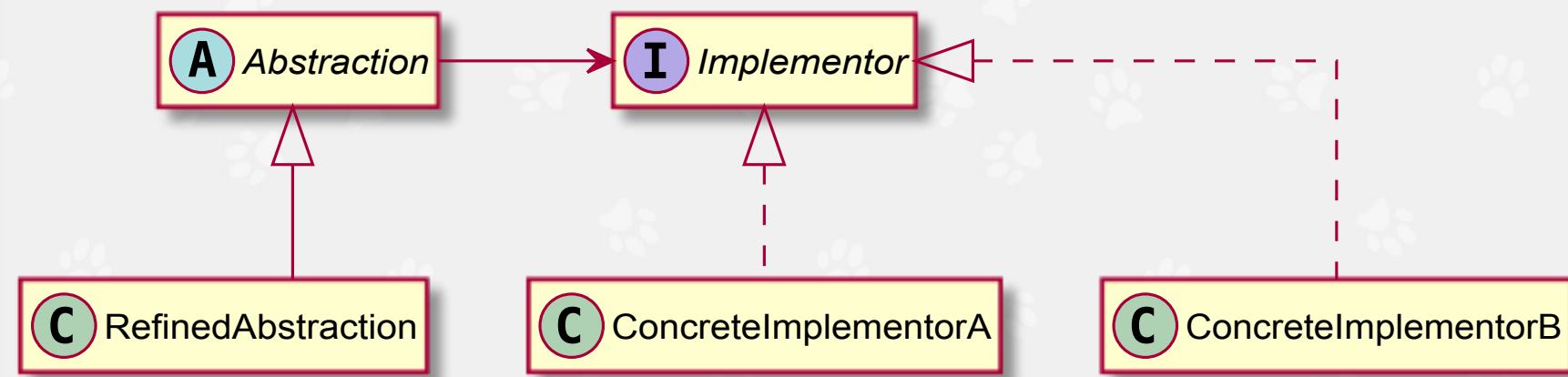
ABSICHT

Decouple an abstraction from its implementation so that the two can vary independently.

— *Gang of Four* 

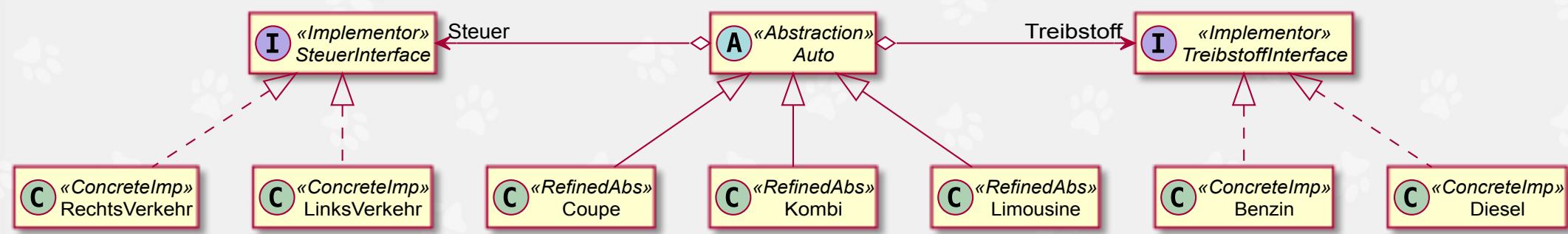
- ❖ Orthogonale Eigenschaften als solche implementieren
- ❖ Mehrfachimplementierung eindämmen
- ❖ Anzahl der Klassen minimieren

STRUKTUR



- ❖ Ausprägungen einer Eigenschaft unter Interface zusammenfassen
- ❖ Subject per Referenz an Eigenschaft binden

STRUKTURBEISPIEL



CHECKLISTE ZUR IMPLEMENTIERUNG

1. Identifiziere das **Subject** und dessen Spezialisierungen.
2. Identifiziere die **Eigenschaft** und deren Ausprägungen.
3. Implementiere das **Interface** der Eigenschaft.
4. Implementiere die Ausprägungen der Eigenschaft.
5. Umstellung des Subjects auf das Interface.

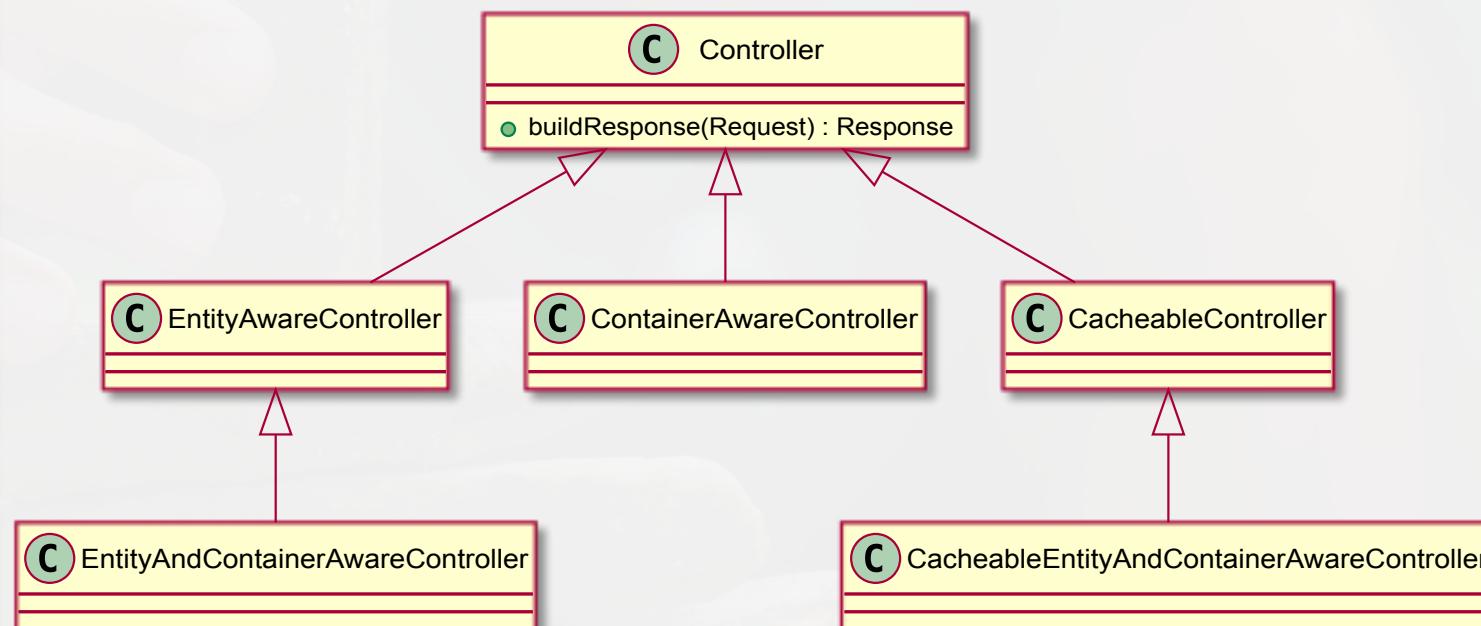
FOLGEN

- Reduziert die Anzahl der Klassen
- Fördert lose Kopplung
- Macht Klassen zur Laufzeit konfigurierbar
- Sorgt für saubere Vererbungshierarchie
- "Separation of concerns"
- Instanziierung u.U. etwas komplexer

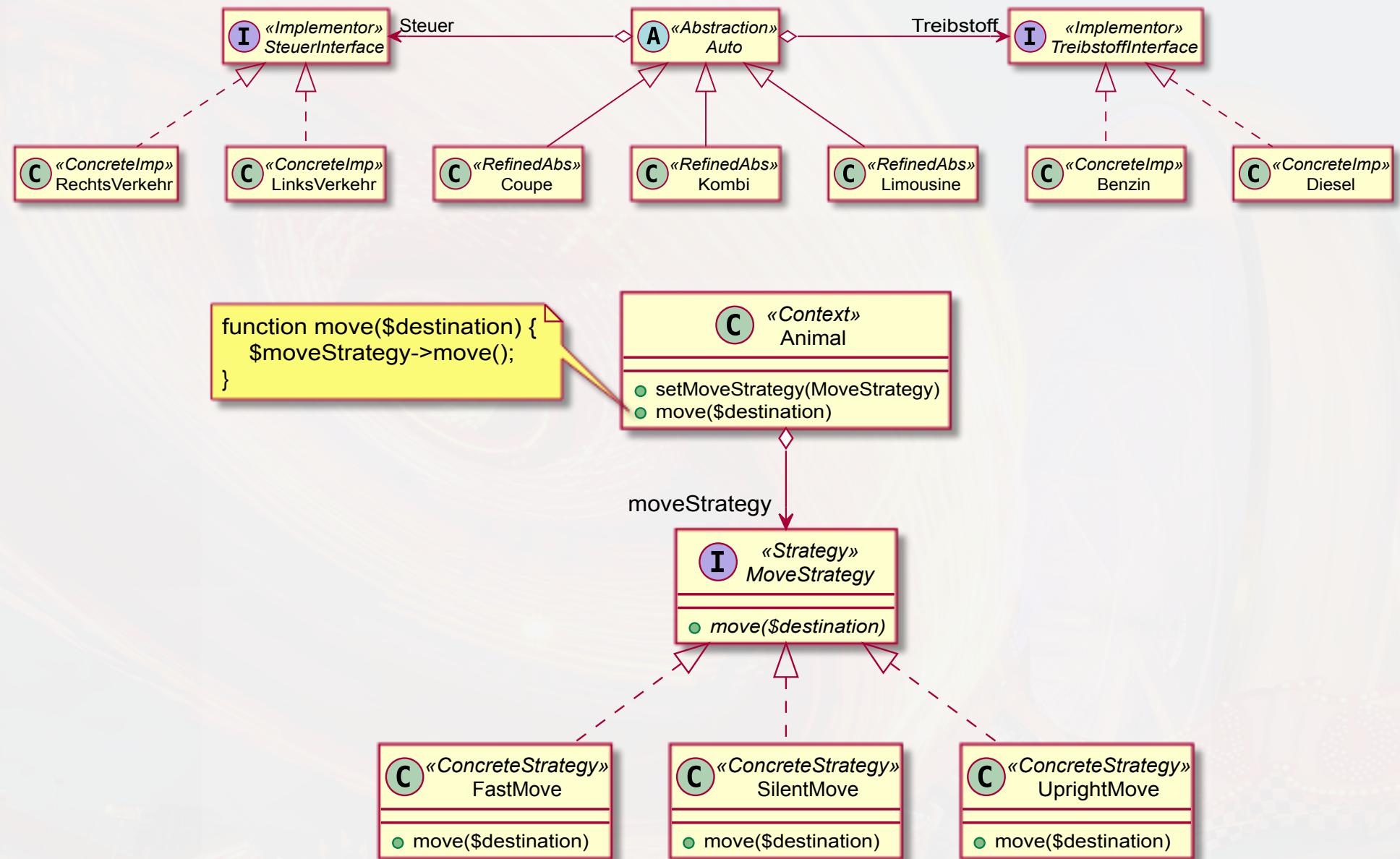
M T W T F S S

1						
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31					

HANDS-ON



BONUSRUNDE BRIDGE + STRATEGY



STATE-PATTERN

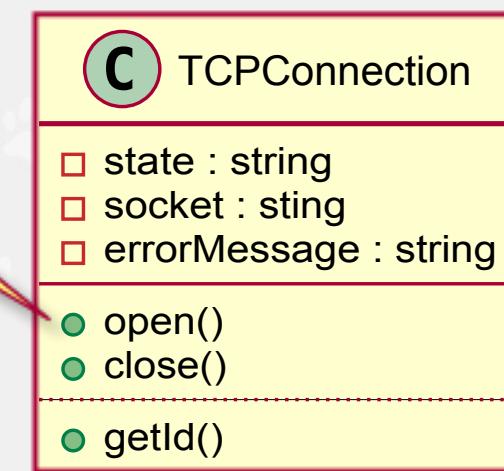


PROBLEMSTELLUNG

Der interne State eines Objekts hängt von verschiedenen privaten Feldern ab und erzeugt bei der Prüfung Overhead und Fehlerquellen.

PROBLEMBEISPIEL

```
if (state === 'open'  
    && socket !== null  
    && errorMessage === null)  
    throw Exception ('Already open');  
  
elseif (state === 'closed'  
       && socket === null  
       && errorMessage === null)  
    socket = open(socket);  
    state = 'open';  
  
elseif (state === 'closed'  
       && socket !== null  
       && errorMessage === null)  
    socket = open(socket);  
    state = 'open';  
[...]  
elseif (state === 'error'  
       && socket !== null  
       && errorMessage !== null)  
    throw Exception (errorMessage);  
)
```

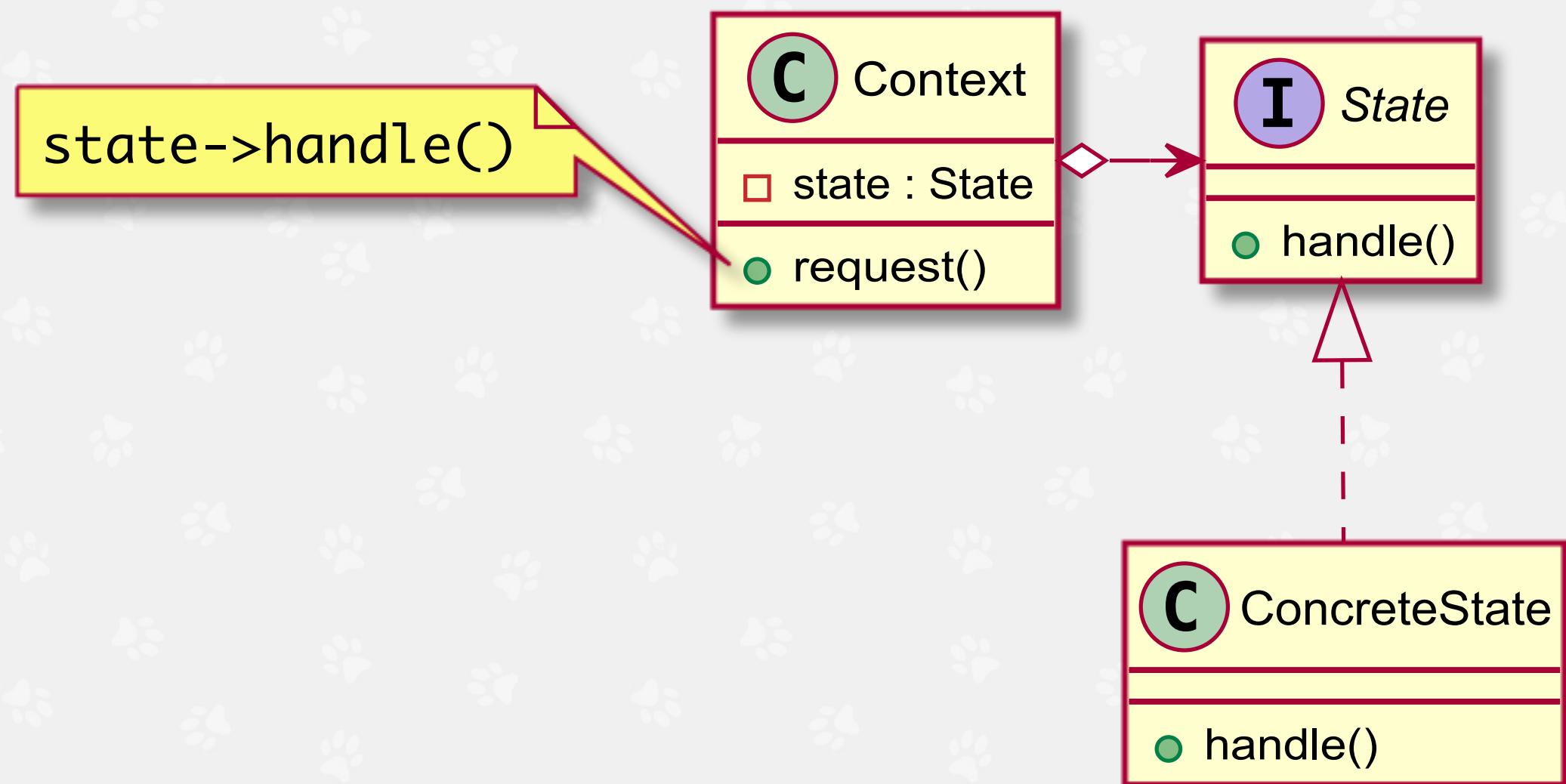


ABSICHT

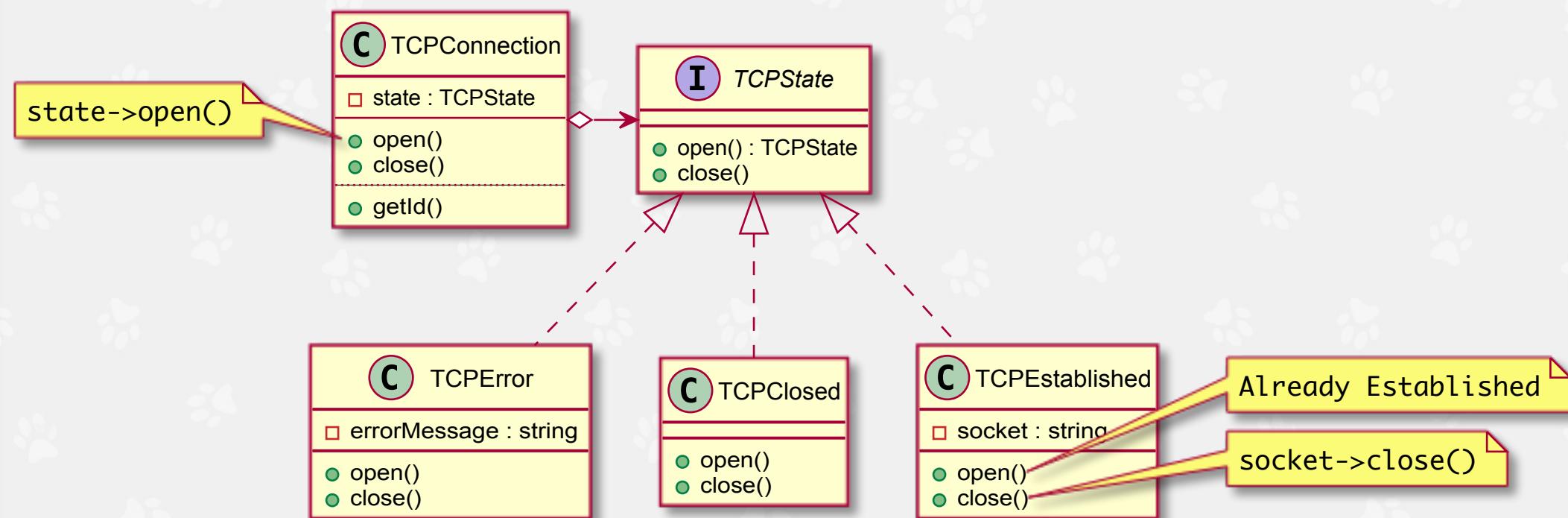
*Allow an object to alter its behavior when its internal state changes.
The object will appear to change its class.*

— *Gang of Four* ↗

STRUKTUR



STRUKTURBEISPIEL



CHECKLISTE ZUR IMPLEMENTIERUNG

1. Identifiziere die **Eigenschaften**, die jeden State beschreiben.
2. Isoliere State-abhängige Methoden in einem **Interface**
3. Erzeugen der **ConcreteState-Klassen**
4. Umschreiben der State-abhängige Methoden im **Context**.

FOLGEN

- Verständlicher Code
 - Code ist weniger fehleranfällig
- Konsistente Zustandsübergänge
- "Separation of concerns"
- Wiederverwendbarkeit von States

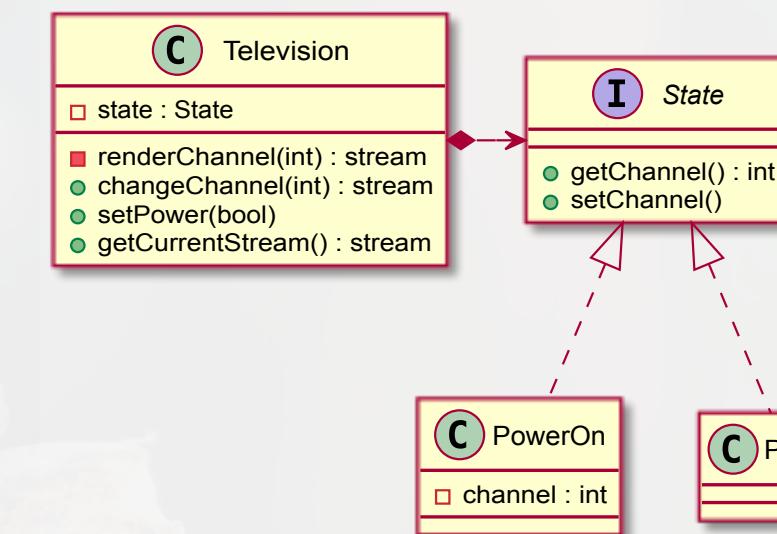
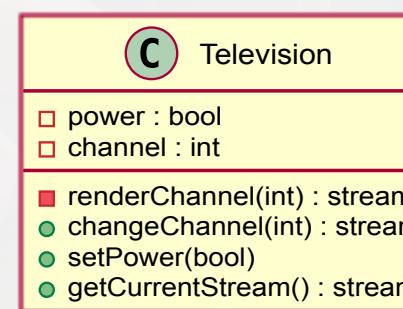


M T W T F S S

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	
16	17	18	19	20	21	22	
23	24	25	26	27	28	29	
30	31						

HANDS-ON

Modelliere den Zustand eines Fernsehers



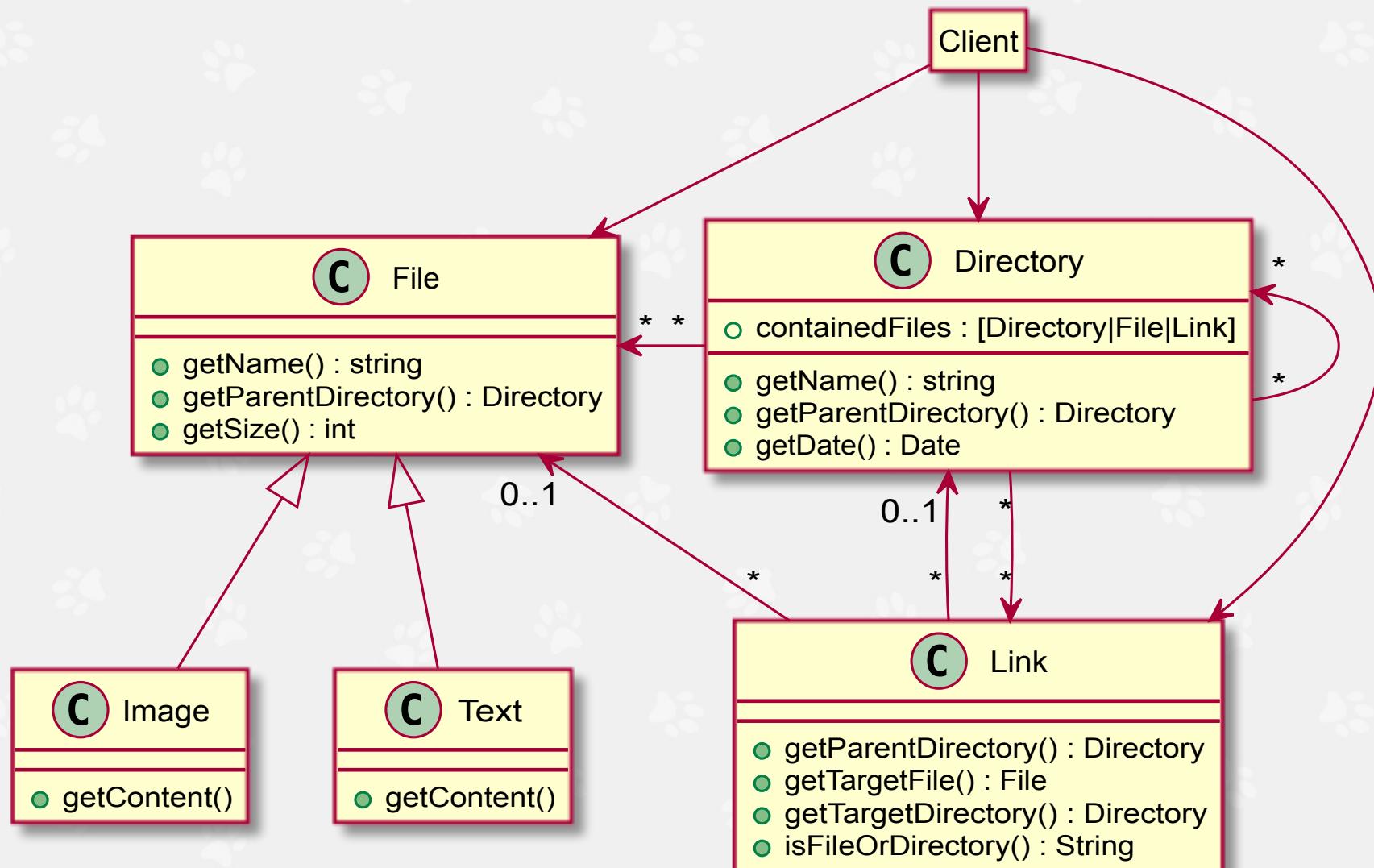
COMPOSITE-PATTERN



PROBLEMSTELLUNG

Eine Menge Baumförmige Struktur von ähnlichen Objekten soll in Klassen modelliert werden — möglichst mit minimalem Implementierungsaufwand.

PROBLEM BEISPIEL



ABSICHT

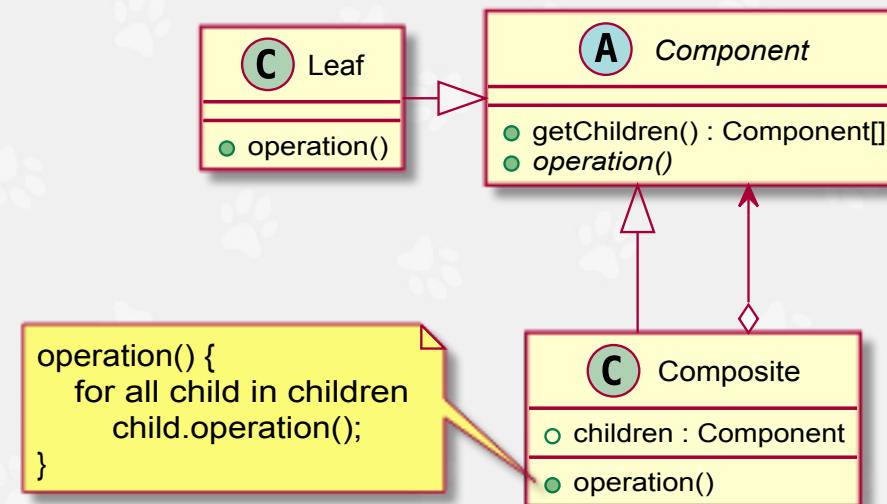
Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

— *Gang of Four* ↗

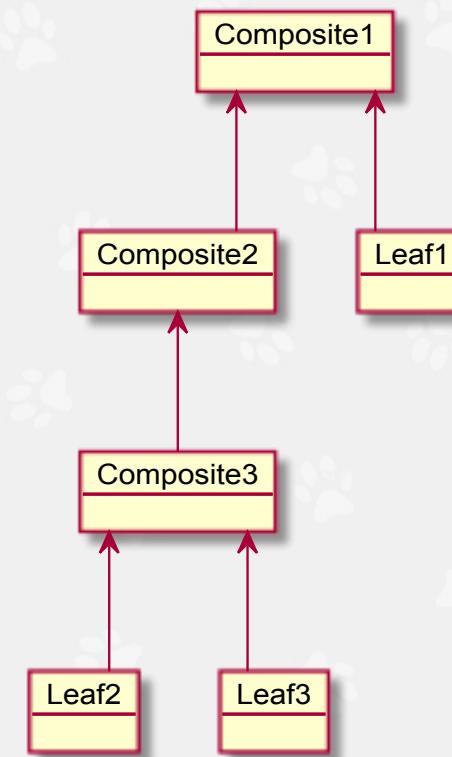
- „ Einheitliche Struktur
 - „ ... für Methoden
 - „ ... für Klassen
- „ Vererbungshierarchie nutzen
- „ Implementierung gegen Interface

STRUKTUR

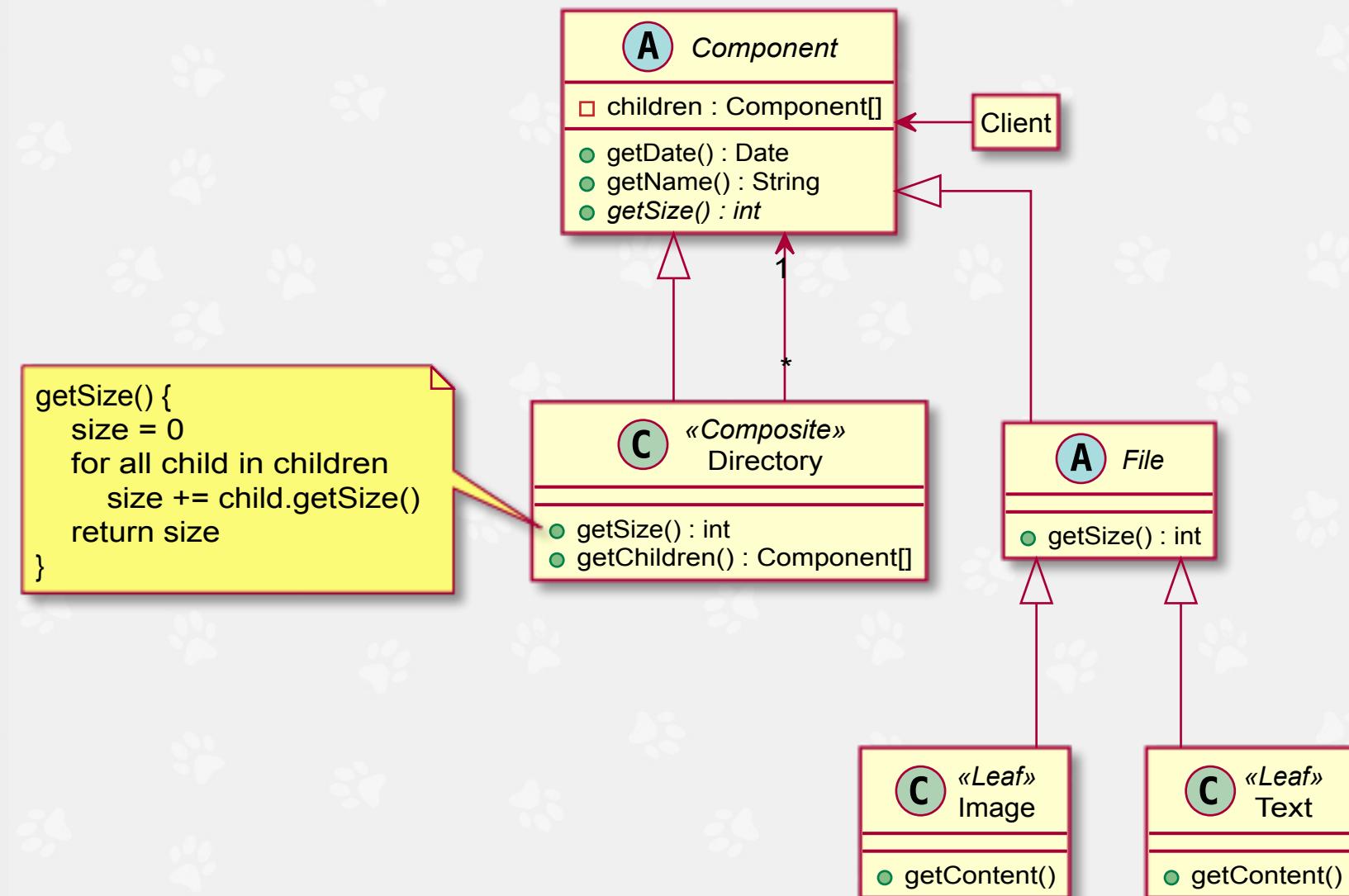
KLASSENDIAGRAMM



OBJEKTDIAGRAMM



STRUKTURBEISPIEL



CHECKLISTE ZUR IMPLEMENTIERUNG

1. Hierarchische Struktur sicher stellen
2. Baumförmige Struktur sicher stellen — innere Knoten und Blätter identifizierbar
3. Gemeinsames Interface für innere Knoten und Blätter definieren
4. Knoten und Blätter auf das Interface umstellen
5. Allen inneren Knoten Relationen zu Objekten des Typs des Interfaces ermöglichen
6. Allen inneren Knoten nutzen in ihren Methoden, die sie aus dem Interface übernommen haben, die Methoden ihrer Kinder

FOLGEN

PRO

- ❖ Komplexe Strukturen durch primitive Klassen abbildbar
- ❖ Wenig Aufwand für Client
- ❖ Einheitliches Interface
- ❖ Leicht erweiterbar

CONTRA

- ❖ Rekursive Operationen ungewohnt
- ❖ Probleme bei iterativen Algorithmen



HANDS-ON

C Vorstand

- getName()
- getUntergebene()

C Bereichsleiter

- getName()
- getUntergebene()

C Abteilungsleiter

- getName()
- getUntergebene()

C Teamleiter

- getName()
- getUntergebene()

C Angestellter

- getName()
- getUntergebene()

❖ Jeder Mitarbeiter soll die Menge aller Untergebenen zurückgeben

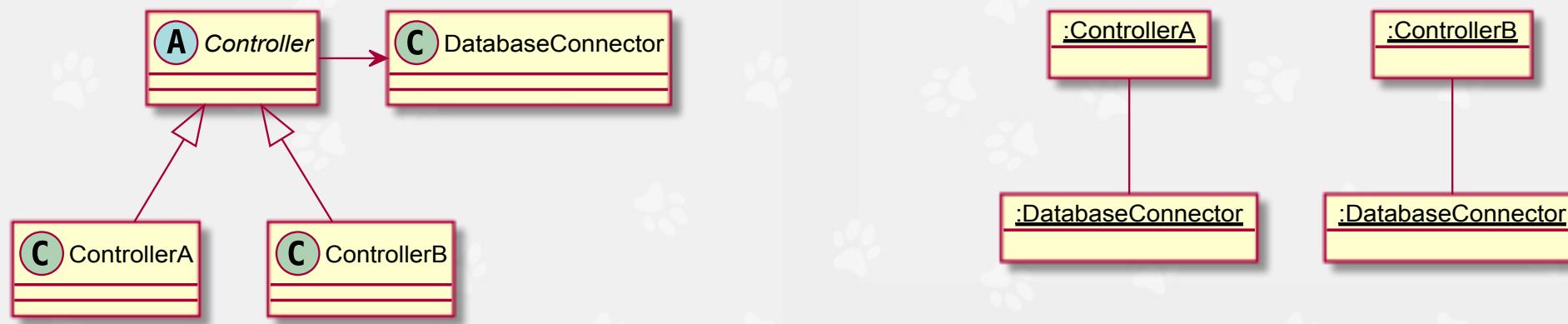
OBJECT POOL-PATTERN



PROBLEMSTELLUNG

Die Menge an Instanzen einer Menge von Klassen ist zu hoch. Die Instanzen sollen an vielen Stellen eines Programms wiederverwendet werden. Man benötigt ein zentrales Verzeichnis.

PROBLEMBEISPIEL

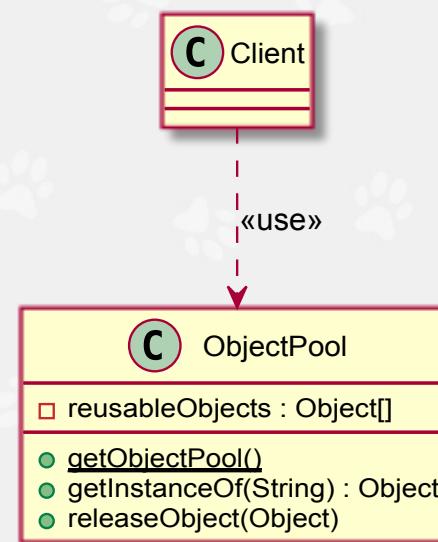


- ❖ Der State des DatabaseConnectors ist wichtig
- ❖ Controller nutzen aber verschiedene Instanzen
- ❖ Mehr Aufwand durch mehrfaches Instanziieren

ABSICHT

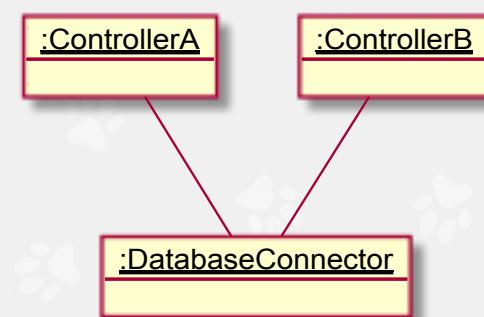
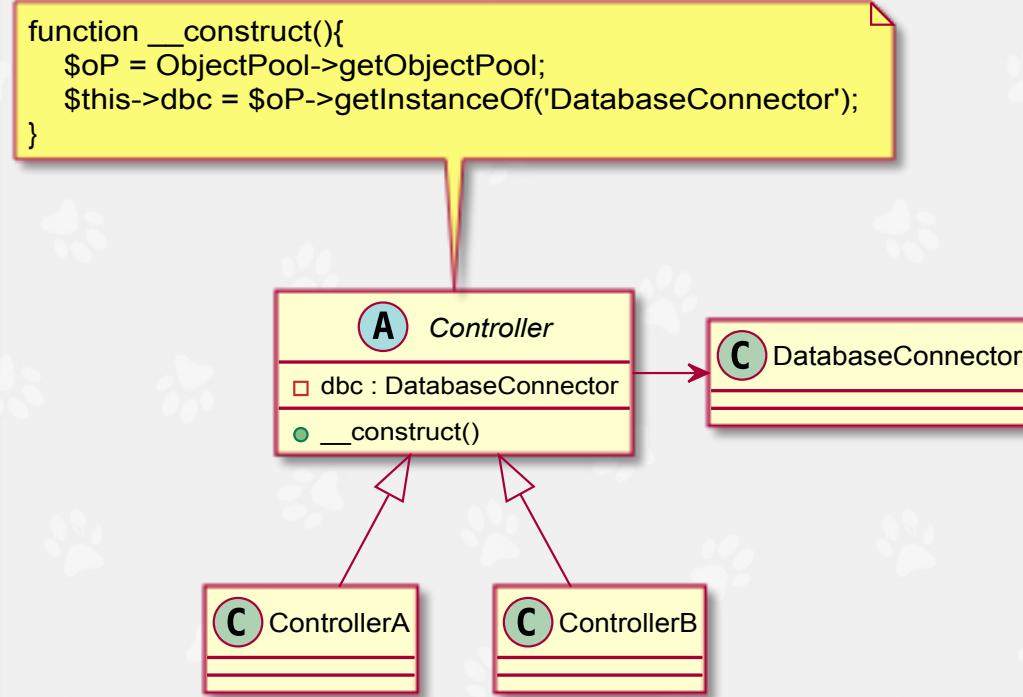
- ❖ Instanzen werden nur so oft erzeugt wie nötig
- ❖ Ein zentraler Dienst verwaltet Instanziierung
- ❖ Zentrales Verzeichnis
- ❖ State bleibt erhalten
- ❖ Ersatz für Singleton Anti-Pattern

STRUKTUR



- ObjectPool ist ein Singleton
- ObjectPool stellt Instanz einer Klasse zur Verfügung
- ObjectPool zerstört auf Anfrage eine Instanz

STRUKTURBEISPIEL



ANTI-PATTERN



STATISCHER SERVICE LOCATOR

DESIGN PATTERN ≠ ROLLE

Eine Rolle beschreibt zu welchem Zweck man ein bestimmtes Design Pattern einsetzt.

CODEBEISPIEL

```
function doSomething() {  
    $databaseConnector = ServiceLocator::getInstance(DatabaseConnector::class);  
    $databaseConnector->select('foo');  
}
```

- ❖ ServiceLocator wird statisch aufgerufen
- ❖ Harte Abhängigkeit
- ❖ Bricht **Dependency Injection** ↪
- ❖ Behindert Tests

CHECKLISTE ZUR IMPLEMENTIERUNG

1. Erzeuge ObjectPool mit privatem Feld vom Typ stdClass []
2. Implementieren der Methoden zum beziehen und löschen von Objekten.
3. Sicherstellen, dass im ganzen Programm nur **ein und die selbe Instanz** von ObjectPool genutzt wird

FOLGEN

- ❖ Je nach Architektur deutlich weniger Instanziierungen von Klassen notwendig
- ❖ Eine zentrale Verwaltung von Instanzen (Vereinfacht Tests)
- ❖ Reusables behalten u.U. ihren State
- ❖ Größe des ObjectPool muss beachtet werden, da Instanzen durch den Garbage Collector nicht mehr abgebaut werden
- ❖ ObjectPool macht Singletons überflüssig

TEMPLATE METHOD-PATTERN



PROBLEMSTELLUNG

Der allergrößte Teil eines Algorithmus bleibt in allen Fällen gleich. Nur ein kleiner Teil muss für verschiedene Anwendungsfälle variiert werden.

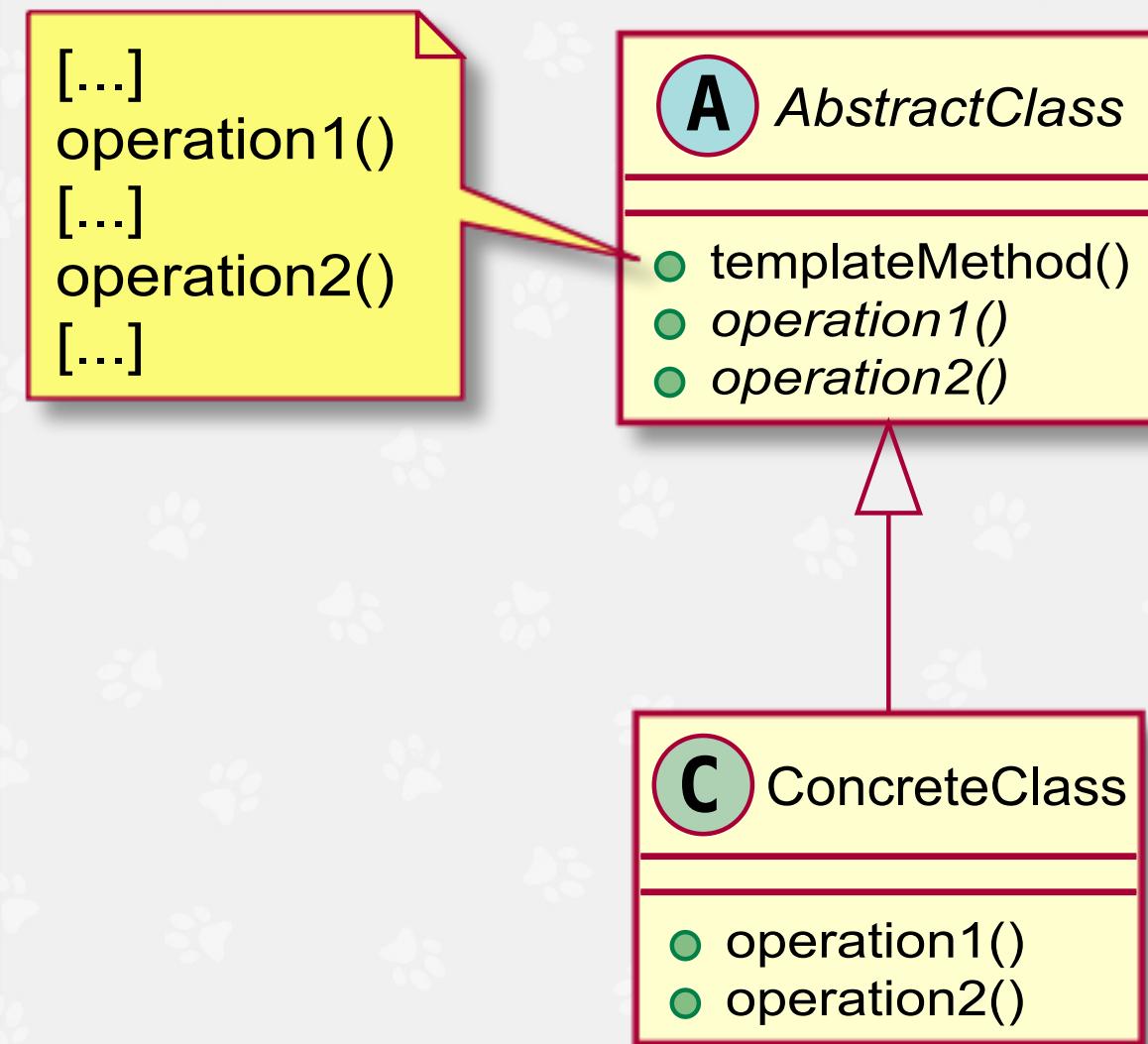
ABSICHT

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

— *Gang of Four* 

- Eine Oberklasse definiert Platzhalter, welche von Unterklassen implementiert werden können

STRUKTUR



- AbstractClass verwaltet den Ablauf
- ConcreteClass implementiert die Arbeitsschritte

STRUKTURBEISPIEL

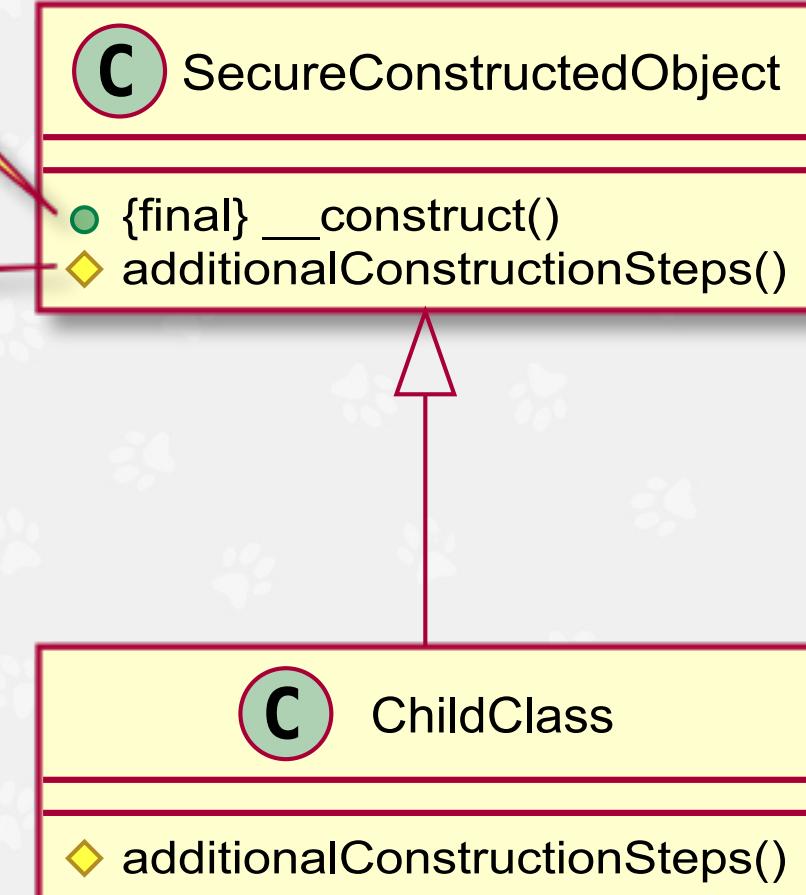
VARIANTEN

STRUKTURBEISPIEL

SICHERHEIT

```
funciton __construct() {  
    $this->highlyImportant();  
    $this->additionalConstructionSteps();  
}
```

empty

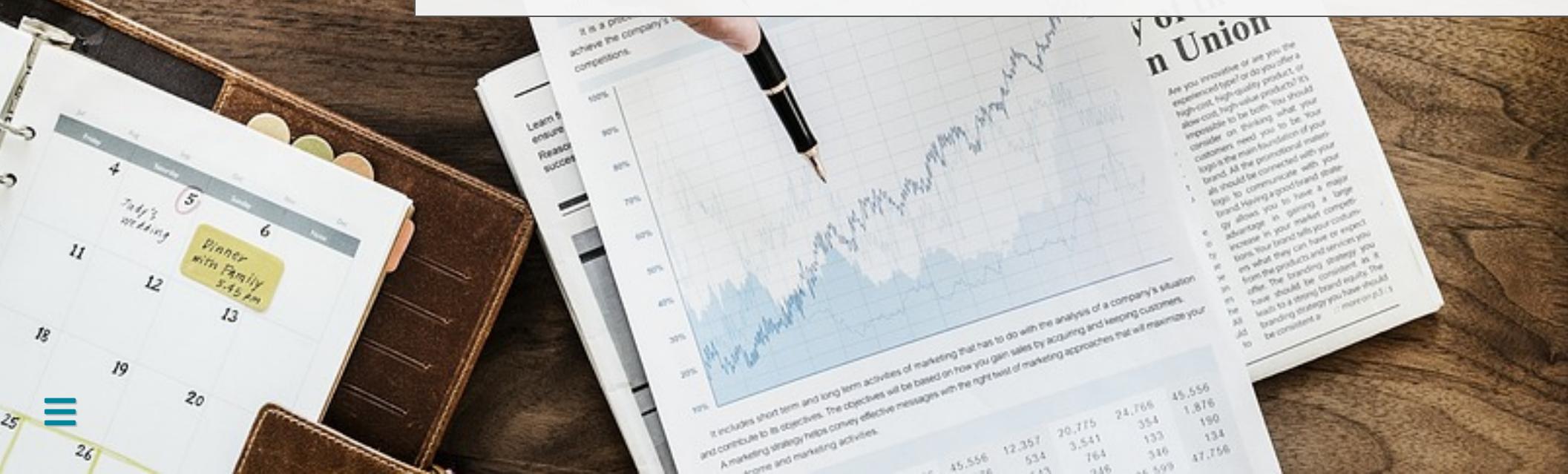


CHECKLISTE ZUR IMPLEMENTIERUNG

1. Den **Algorithmus in Operationen unterteilen**, die standardmäßig oder nur in speziellen Situationen ausgeführt werden sollen
2. **Definition einer abstrakten Basisklasse**, welche die Standardteile in Methoden implementiert
3. **Definition einer Platzhaltermethode** für spezielles Verhalten
4. Rufe die Platzhaltermethoden in der abstrakten Klasse auf
5. **Deklariere beliebig viele Kindklassen**, die das spezielle Verhalten in ihren Methoden implementieren

FOLGEN

- ❖ Gut zu erweitern
- ❖ Hoher Grad an Wiederverwendbarkeit
- ❖ Möglichkeit Programmablauf festzulegen
- ❖ Inversion of Control (Don't call us, we'll call you)



	M	T	W	T	F	S	S
1							
2	2	3	4	5	6	7	8
9	9	10	11	12	13	14	15
16	16	17	18	19	20	21	22
23	23	24	25	26	27	28	29
30	30	31					

HANDS-ON AUFGABE

- ❖ Implementiere einen Prozess mit den Schritten "Setup", "Execute", "TearDown".
- ❖ Die drei Phasen werden **der Reihe nach** durchlaufen.
- ❖ Es soll möglichst einfach sein **Varianten zu Bilden**, die eigene Logik für die drei Phasen implementieren sollen



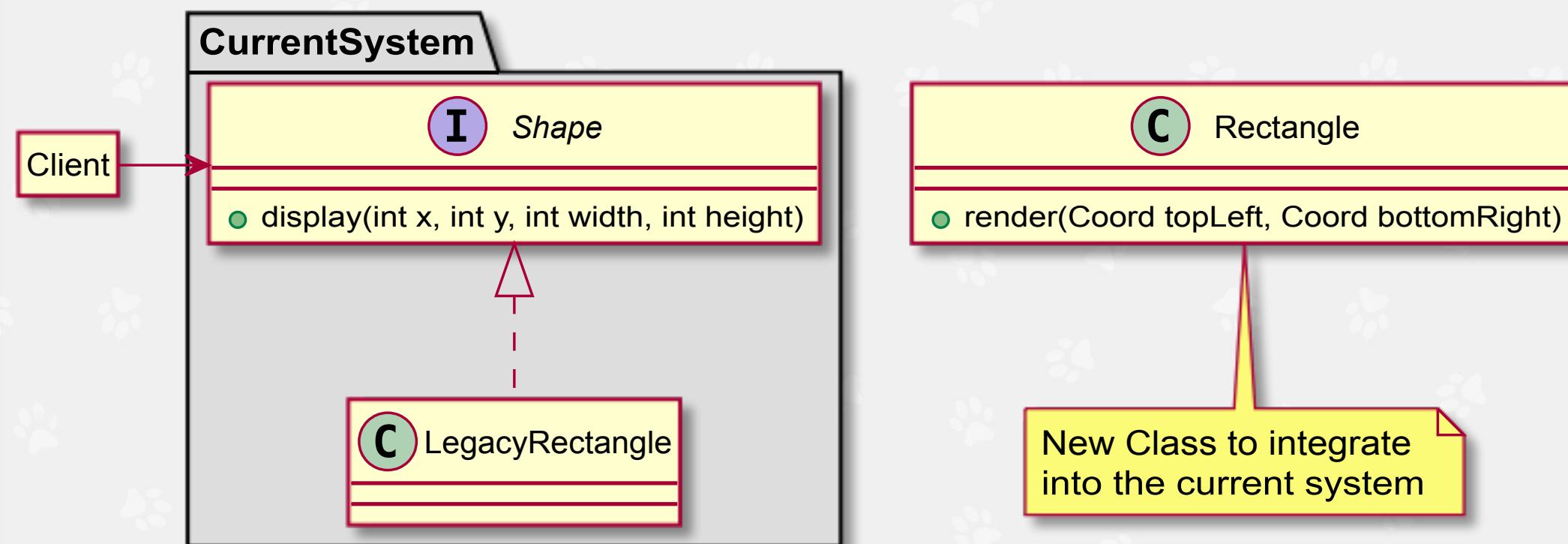
ADAPTER-PATTERN



PROBLEMSTELLUNG

Zwei vorhandene Klassenstrukturen passen nicht zueinander, aber eine der beiden neu zu implementieren ist zu aufwendig.

PROBLEMBEISPIEL



ABSICHT

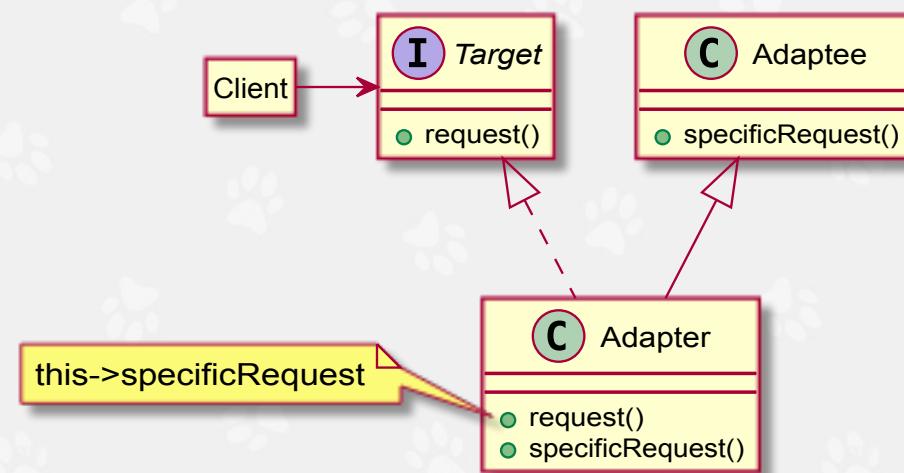
*Convert the interface of a class into another interface clients expect.
Adapter lets classes work together that couldn't otherwise because of
incompatible interfaces.*

— *Gang of Four* 

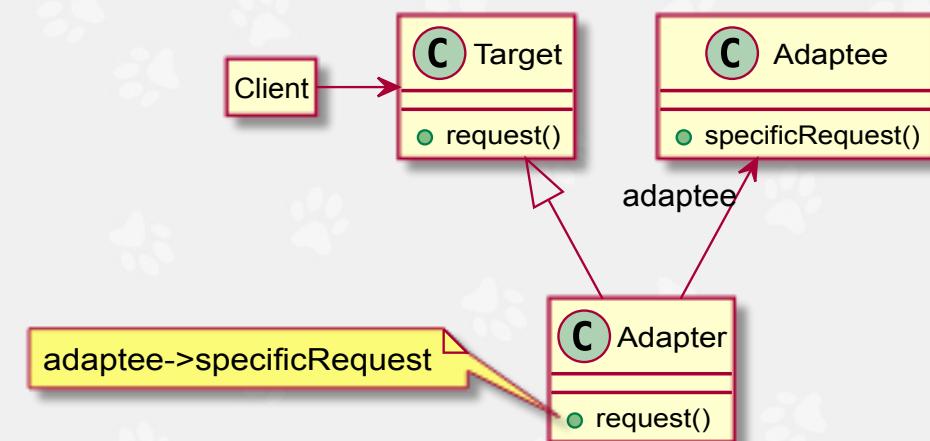
STRUKTUR

Es gibt zwei Herangehensweisen an dieses Pattern.

VERERBUNG



REFERENZ

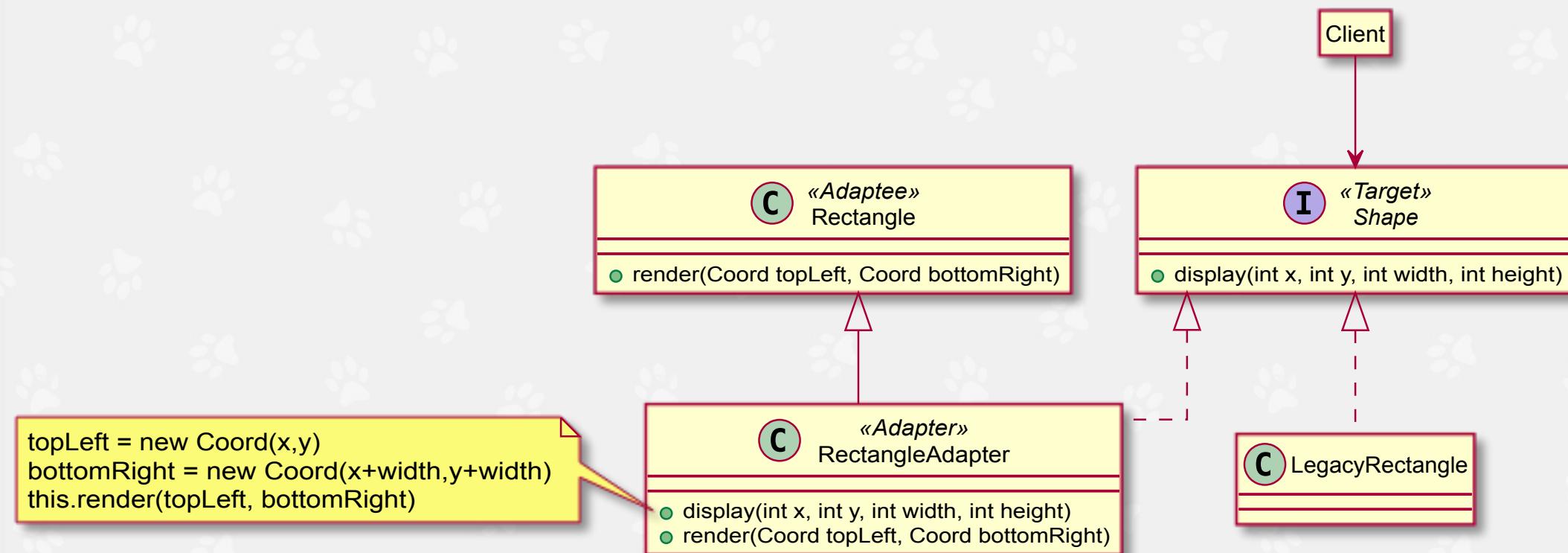


- Adapter erbt von Interface und neuer Klasse
- Erfordert Interface

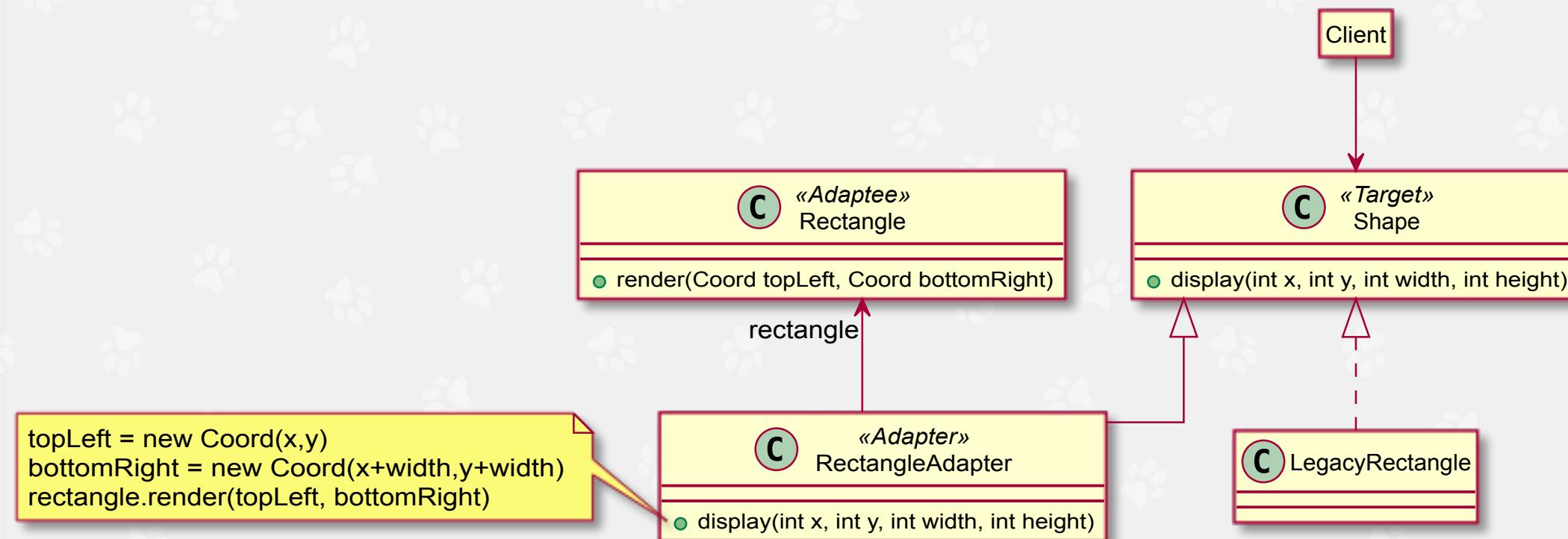
- Adapter hat Referenz auf Adaptee

STRUKTURBEISPIEL

VERERBUNG



REFERENZ



CHECKLISTE ZUR IMPLEMENTIERUNG

1. Identifiziere Target und Adaptee
2. Wähle Patternvariante
3. Erzeuge Adapter
4. Implementiere Methoden des Adapters

FOLGEN

VERERBUNG

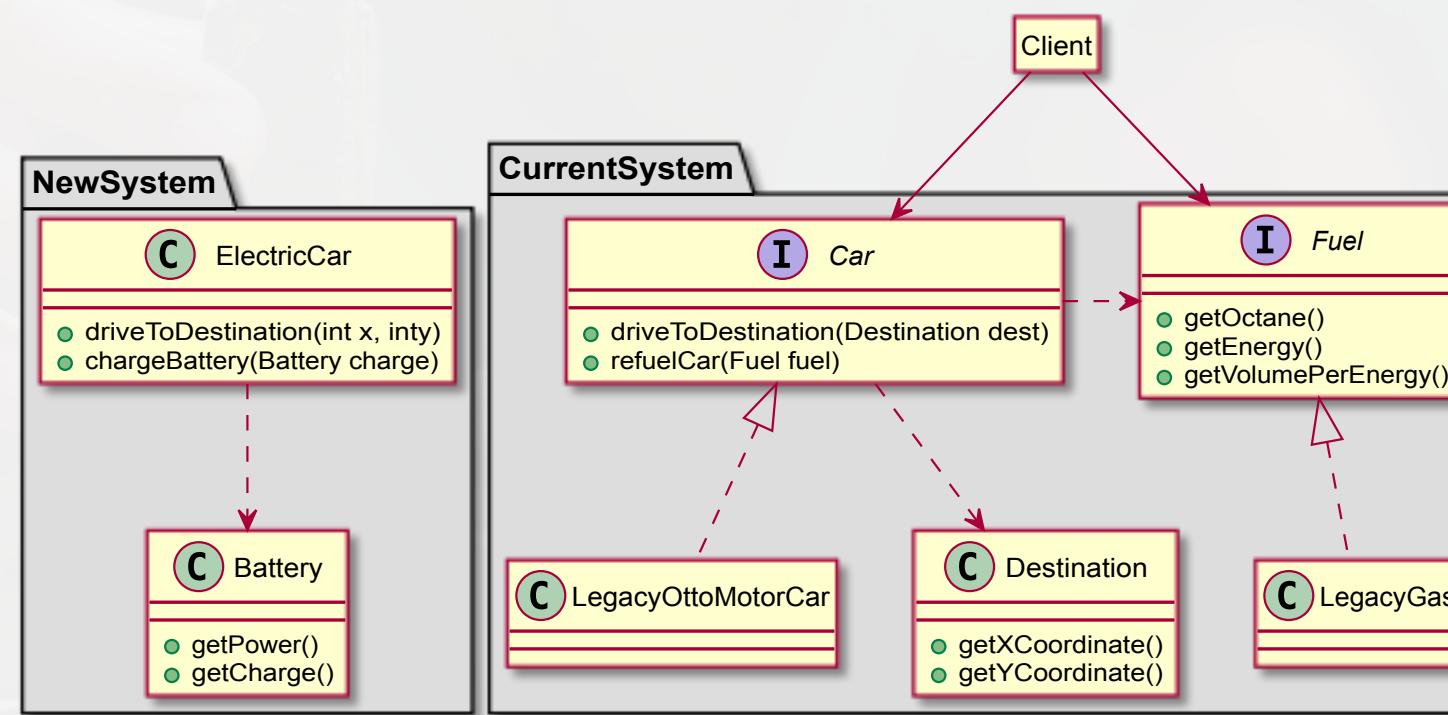
- Adapter funktioniert nicht für Kinder von Adaptee
- Adapter kann Funktionalität von Adaptee verändern
- Nur eine Instanz nötig
- Vermischt alte und neue Klassenkonzepte

REFERENZ

- Ein Adapter kann prinzipiell alle Kinder von Adaptee anbinden
- Kann Funktionalität ergänzen — Nicht überschreiben



HANDS-ON



FAZIT

PATTERN

Adapter	State
Template	Bridge
Object Pool	Facade
Composite	

ZWECK

- Lösungen skizzieren
 - Dokumentation
 - Gemeinsame Sprache
 - Saubere Architektur



leichen
Danke!

Exkurs:

ANTI-PATTERN

An anti-pattern is a common *response to a recurring problem* that is usually *ineffective* and risks being *highly counterproductive*.

— Software design. Harlow, Eng.: Addison-Wesley

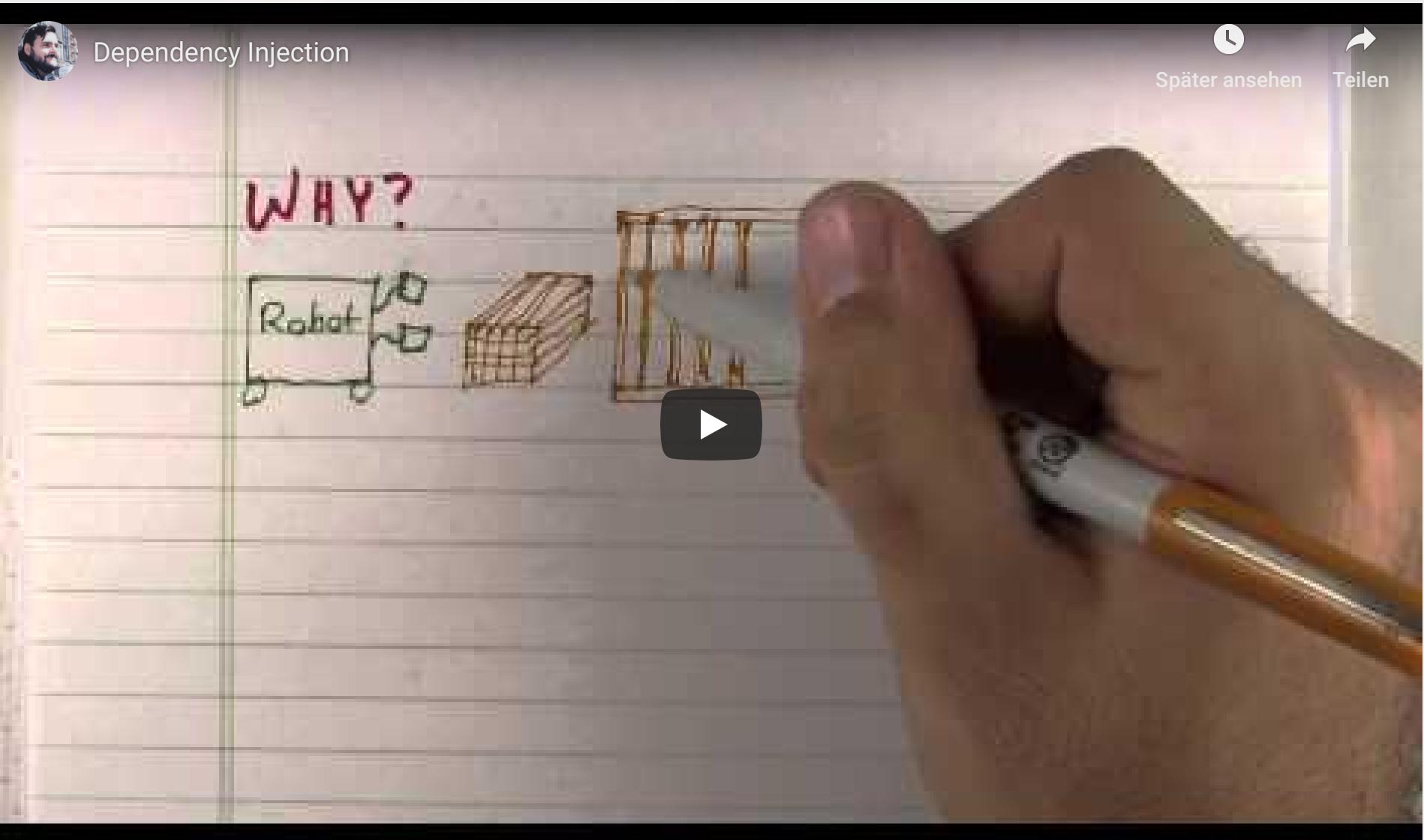
BEKANNTE ANTI-PATTERN

Software Development	Software Architektur	Project Management
The God Class	Vendor Lock-in	Fear of Success
Golden Hammer	Warm Bodies	Death By Planning
Spaghetti Code	Reinvent The Wheel	Intellectual Violence

 Zurück zum Thema 

Exkurs:

DEPEDENCY-INJECTION



 Zurück zum Thema 