Software Engineering II

9 Entwurfsmuster

SS 2020

Prof. Dr. Dirk Müller

Übersicht

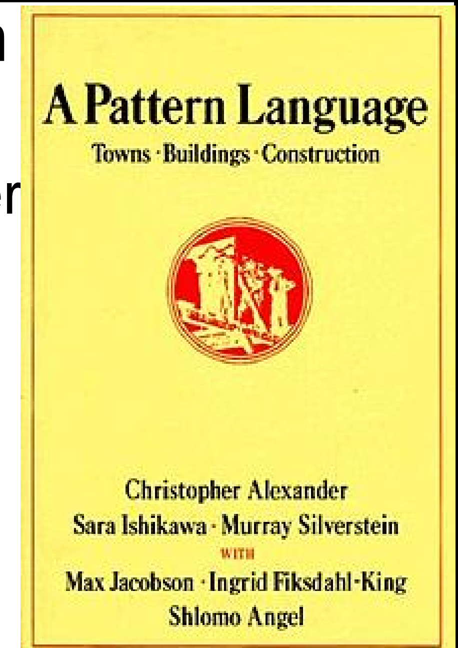
- Einführung
- Geschichte
- Anforderungen und Nutzen
- Grundlegende Elemente
- Alphabetischer GoF-Katalog
- 11 GoF-Muster im Detail
- Nachteile
- Zusammenfassung

Einführung

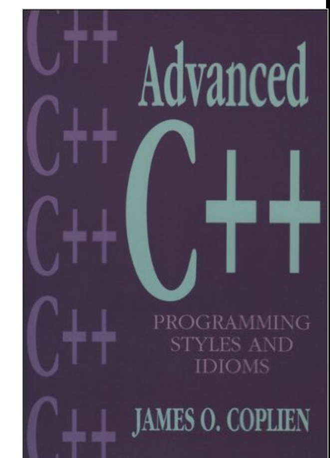
- Entwurfsmuster sind typischerweise in Anwendungen vorkommende Kombinationen von Klassen/Objekten zu größeren Einheiten.
  - aus Erfahrungen abstrahiert
  - zur Nachnutzung bestimmt
- zunächst unabhängig von der Programmiersprache
- Gamma *et al.* haben einen Katalog von 23 Entwurfsmustern beschrieben. [2]
  - alphabetische Auflistung und Klassifikation
  - Fokus auf objektorientierten Ansatz mit UML-Diagrammen
- Anwendungsbeispiel: MVC-Architektur
  - Verständnis, welche Muster eingesetzt werden

Geschichte

- 1977 *Christopher Alexander*: Sammlung von Entwurfsmustern in der Architektur [3]
  - Idee der frühzeitigen Einbeziehung der Bewohner
  - 253 (!) Muster auf 1171 Seiten beschrieben
  - enthält bereits hypertextartige Verweise

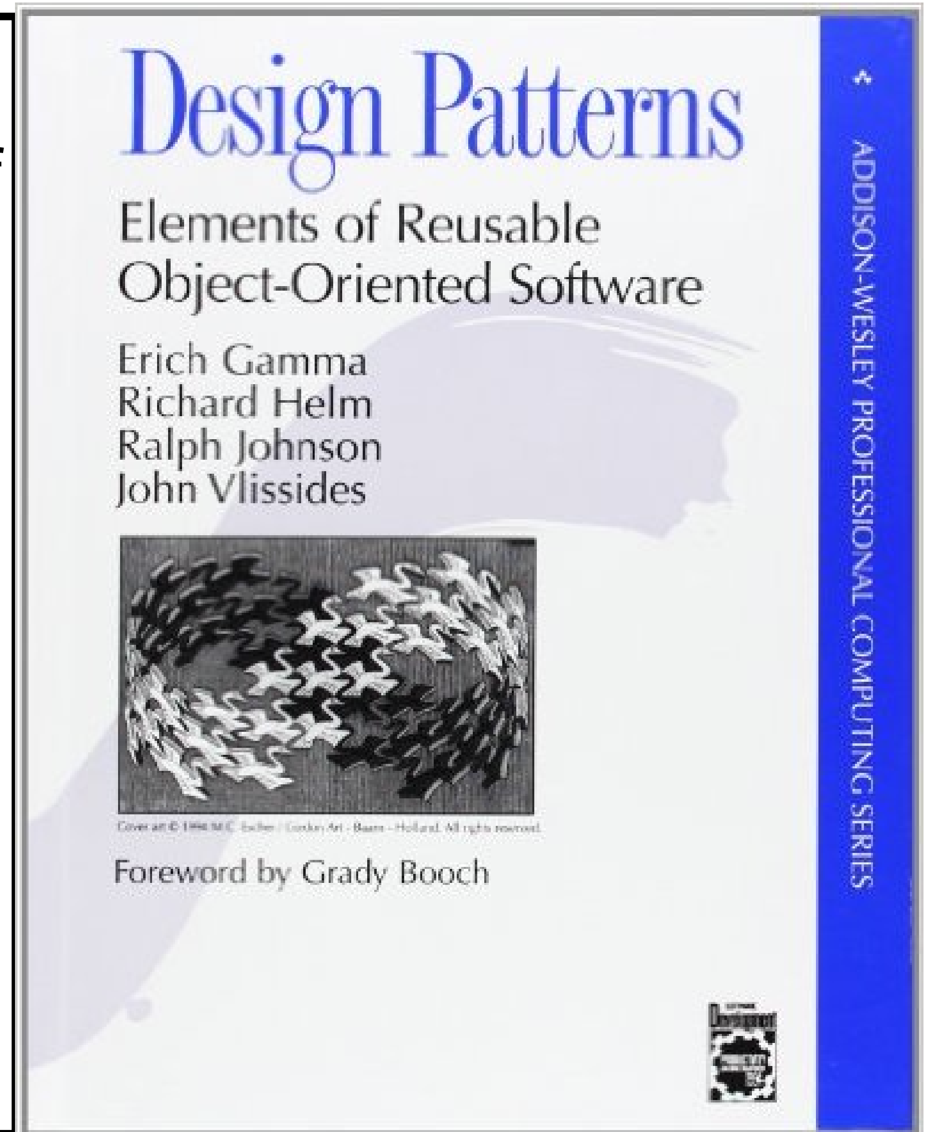


- 1987 *Kent Beck* und *Ward Cunningham*: Muster für grafische Benutzeroberflächen in *Smalltalk*
- 1991 *James Coplien*: Idiome als programmiersprachenspezifische Muster auf einer niedrigen Abstraktionsebene, hier für *C++* [4]
  - Sichtweise: Umsetzung eines Musters mit den Mitteln einer konkreten Programmiersprache



Viererbande

- 1994 Erich Gamma *et al.*: „Design Patterns – Elements of Reusable Object-Oriented Software“ [2]
- Titel zu lang, um ihn in einer E-Mail zu zitieren => Abkürzung zu *Gang of Four* (GoF) bzw. Viererbande
  - sowohl für Autorenteam als auch für dieses Standardwerk genutzt
- Musterkatalog
  - 23 Entwurfsmuster in 3 Kategorien eingeordnet



Quelle: [2]

Anforderungen an ein Entwurfsmuster

- Lösung eines oder mehrerer Probleme
- Praxistauglichkeit
  - bietet erprobtes Konzept
  - basiert auf realen Entwürfen
- Nichttrivialität
  - geht über das rein Offensichtliche hinaus
  - Beziehungen aufzeigen, die tiefer gehende Strukturen und Mechanismen eines Systems umfassen
- Einbindung des Benutzers in den Entwurfsprozess
- Katalogtauglichkeit
  - kurze, prägnante Benennung; erleichtert Verweis in Quelltexten oder Diskussionen
  - enthält Verweise auf andere Muster; ermöglicht so den Aufbau von Mustersprachen

Nutzen

- bewährte Lösung für bestimmte Klasse von Entwurfsproblemen („*Best Practice*")
- kurze eindeutige Benennung erleichtert Verweise in Quelltexten und Diskussionen
  - unabhängig von Programmiersprache
- bei Verweis auf Muster aus Musterkatalogen (z. B. in Quelltexten) dortige Diskussion des Problemkontextes und der Vor- und Nachteile sowie möglicher Alternativen mit referenziert

Grundlegende Elemente von Mustern

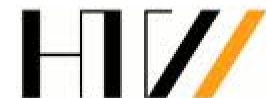
- Name
  - 1-2 Wörter, erweitert das Entwurfsvokabular
- Problem
  - Problem und Anwendungskontext werden erklärt
- Lösung
  - abstrakte Beschreibung eines Entwurfsproblems
  - kein konkreter Entwurf
  - keine konkrete Implementierung
- Folgen
  - Kosten-Nutzen-Verhältnis von Mustern abwägen
  - Ressourcenbedarf (Speicher, Laufzeit)
  - Auswirkungen auf Flexibilität, Erweiterbarkeit, Portabilität

Quelle: [2]

GoF-Katalog

Erzeugungsmuster	Strukturmuster	Verhaltensmuster
• Fabrikmethode klassenbasiert	• Adapter (klassenbasiert)	• Interpreter • Schablonenmethode
• Abstrakte Fabrik • Erbauer • Prototyp • *Singleton* objektbasiert	• Adapter (objektbasiert) • Brücke • Kompositum • Dekorierer • Fassade • Fliegengewicht • Stellvertreter	• Zuständigkeitskette • Kommando • Iterator • Vermittler • Memento • Beobachter • Zustand • Strategie • Besucher

Quelle: [2]



Klassifizierung im GoF-Katalog

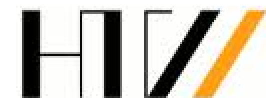
- Erzeugungsmuster
  - Objekterzeugung gekapselt und ausgelagert
  - somit Kontext der Objekterzeugung unabhängig von konkreter Implementierung
  - Regel „Programmiere auf die Schnittstelle, nicht auf die Implementierung!“ wird beachtet
- Strukturmuster
  - Schablonen für die Beziehungen zwischen Klassen bzw. Objekten
- Verhaltensmuster
  - Modellierung komplexen Verhaltens

vertikal

- klassenbasiert
- objektbasiert

horizontal

Quelle: [2]



Alphabetische Liste der GoF-Muster (1/7)

- Abstrakte Fabrik (*Abstract Factory*)
  - Biete eine Schnittstelle zum Erzeugen von Familien verwandter oder voneinander abhängiger Objekte, ihre konkreten Klassen zu benennen.
- Adapter (*Adapter*)
  - Passe die Schnittstelle einer Klasse an eine andere von ihren Klienten erwartete Schnittstelle an. Das Adaptermuster lässt Klassen zusammenarbeiten, die wegen inkompatibler Schnittstellen ansonsten dazu nicht in der Lage wären.
- Befehl (*Command*)
  - Kapsle einen Befehl als ein Objekt. Dies ermöglicht es, Klienten mit verschiedenen Anfragen zu parametrisieren, Operationen in eine *Queue* zu stellen, ein Logbuch zu führen und Operationen rückgängig zu machen.

Quelle: [2]

Alphabetische Liste der GoF-Muster (2/7)

- Beobachter (*Observer*)
  - Definiere eine 1-zu-n-Abhängigkeit zwischen Objekten, so dass die Änderung des Zustands eines Objekts dazu führt, dass alle abhängigen Objekte benachrichtigt und automatisch aktualisiert werden.
- Besucher (*Visitor*)
  - Kapsle eine auf den Elementen einer Objektstruktur auszuführende Operation als ein Objekt. Das Besuchermuster ermöglicht es Ihnen, eine neue Operation zu definieren, ohne die Klassen der von ihr bearbeiteten Elemente zu verändern.
- Brücke (*Bridge*)
  - Entkopple eine Abstraktion von ihrer Implementierung, so dass beide unabhängig voneinander variiert werden können.
- Dekorierer (*Decorator*)
  - Erweitere ein Objekt dynamisch um Zuständigkeiten. Dekorierer bieten eine flexible Alternative zur Unterklassenbildung, um die Funktionalität einer Klasse zu erweitern.

Quelle: [2]

Alphabetische Liste der GoF-Muster (3/7)

- Erbauer (*Builder*)
  - Trenne die Konstruktion eines komplexen Objekts von seiner Repräsentation, so dass derselbe Konstruktionsprozess unterschiedliche Repräsentationen erzeugen kann.
- Fabrikmethode (*Factory Method*)
  - Definiere eine Klassenschnittstelle mit Operationen zum Erzeugen eines Objekts, aber lasse Unterklassen entscheiden, von welcher Klasse das zu erzeugende Objekt ist. Fabrikmethoden ermöglichen es einer Klasse, die Erzeugung von Objekten an Unterklassen zu delegieren.
- Fassade (*Façade*)
  - Biete eine einheitliche Schnittstelle zu einer Menge von Schnittstellen eines Subsystems. Die Fassadenklasse definiert eine abstrakte Schnittstelle, welche die Verwendung des Subsystems vereinfacht.

Quelle: [2]

Alphabetische Liste der GoF-Muster (4/7)

- Fliegengewicht (*Flyweight*)
  - Nutze Objekte kleinster Granularität gemeinsam, um große Mengen von ihnen effizient verwenden zu können.
- Interpreter (*Interpreter*)
  - Definiere für eine gegebene Sprache eine Repräsentation der Grammatik sowie einen Interpreter, der die Repräsentation nutzt, um Sätze in der Sprache zu interpretieren.
- Iterator (*Iterator*)
  - Ermögliche den sequenziellen Zugriff auf die Elemente eines zusammengesetzten Objekts, ohne seine zugrunde liegende Repräsentation offenzulegen.
- Kompositum (*Composite*)
  - Füge Objekte zu Baumstrukturen zusammen, um Teile-Ganzes-Hierarchien zu repräsentieren und Kompositionen von Objekten einheitlich zu behandeln.

Quelle: [2]

Alphabetische Liste der GoF-Muster (5/7)

- Memento (*Memento*)
  - Erfasse und externalisiere den internen Zustand eines Objektes ohne seine Kapselung zu verletzen, so dass das Objekt später in diesen Zustand zurückversetzt werden kann.
- Prototyp (*Prototype*)
  - Bestimme die Arten zu erzeugender Objekte durch die Verwendung eines prototypischen Exemplars, und erzeuge neue Objekte durch Kopieren dieses Prototypen.
- Stellvertreter (*Proxy*)
  - Kontrolliere den Zugriff auf ein Objekt mit Hilfe eines vorgelagerten Stellvertreterobjekts.
- Schablonenmethode (*Template Method*)
  - Definiere das Skelett eines Algorithmus in einer Operation und delegiere einzelne Schritte an Unterklassen. Die Verwendung einer Schablonenmethode ermöglicht es Unterklassen, bestimmte Schritte eines Algorithmus zu überschreiben, ohne seine Struktur zu verändern.

Quelle: [2]

Alphabetische Liste der GoF-Muster (6/7)

- Einzelstück (*Singleton*)
  - Sichere ab, dass eine Klasse genau ein Exemplar besitzt, und stelle einen globalen Zugriffspunkt darauf bereit.
- Strategie (*Strategy*)
  - Definiere eine Familie von Algorithmen, kapsele jeden einzelnen und mache sie austauschbar. Das Strategiemuster ermöglicht es, den Algorithmus unabhängig von ihn nutzenden Klienten zu variieren.
- Vermittler (*Mediator*)
  - Definiere ein Objekt, welches das Zusammenspiel einer Menge von Objekten in sich kapselt. Vermittler fördern lose Kopplung, indem sie Objekte davon abhalten, aufeinander explizit Bezug zu nehmen. Sie ermöglichen es Ihnen, das Zusammenspiel der Objekte von ihnen unabhängig zu variieren.

Quelle: [2]

Alphabetische Liste der GoF-Muster (7/7)

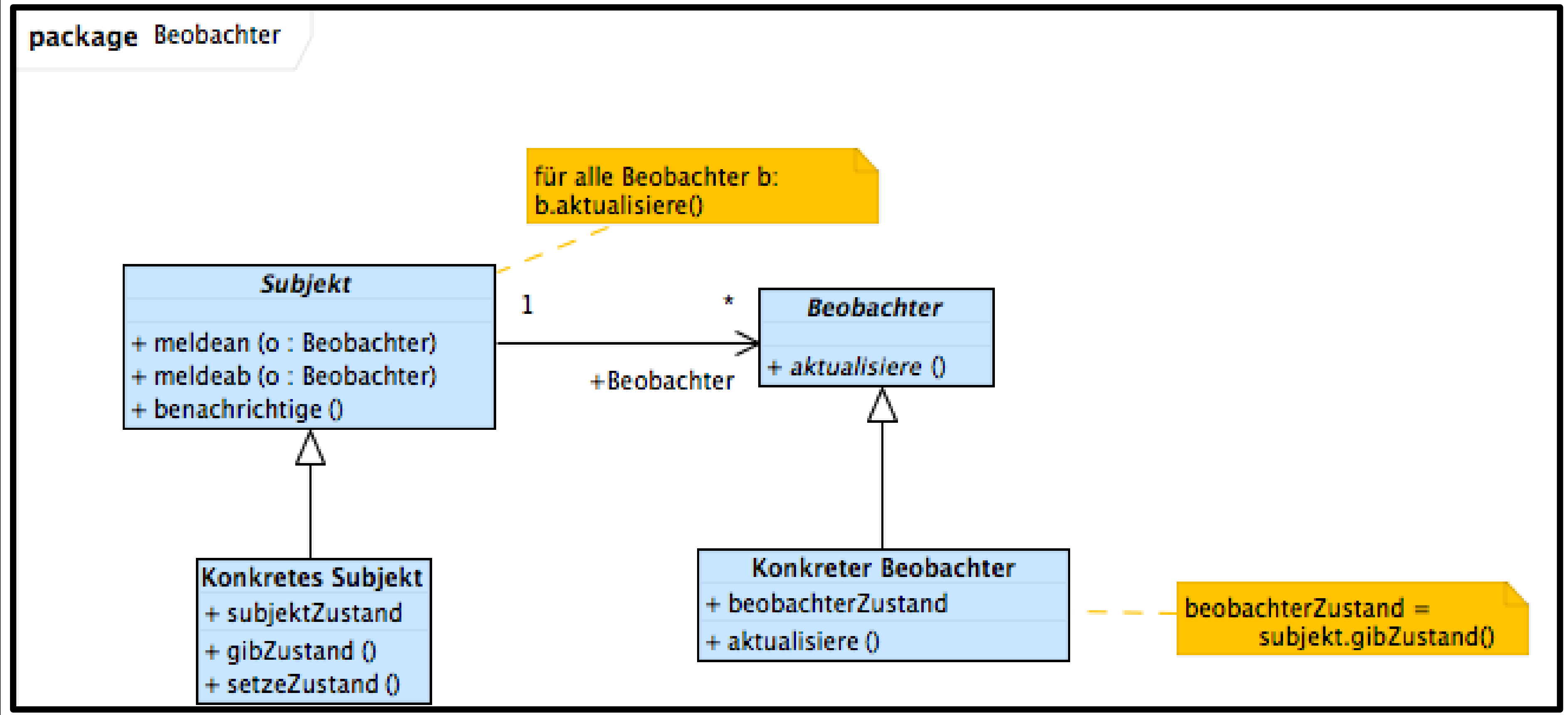
- Zustand (*State*)
  - Ermögliche es einem Objekt, sein Verhalten zu ändern, wenn sein interner Zustand sich ändert. Es wird so aussehen, als ob das Objekt seine Klasse gewechselt hat.
- Zuständigkeitskette (*Chain of Responsibility*)
  - Vermeide die Kopplung eines Auslösers einer Anfrage an seinen Empfänger, indem mehr als ein Objekt die Möglichkeit erhält, die Anfrage zu erledigen. Verkette die empfangenen Objekte, und leite die Anfrage an der Kette entlang, bis ein Objekt sie erledigt.

Quelle: [2]

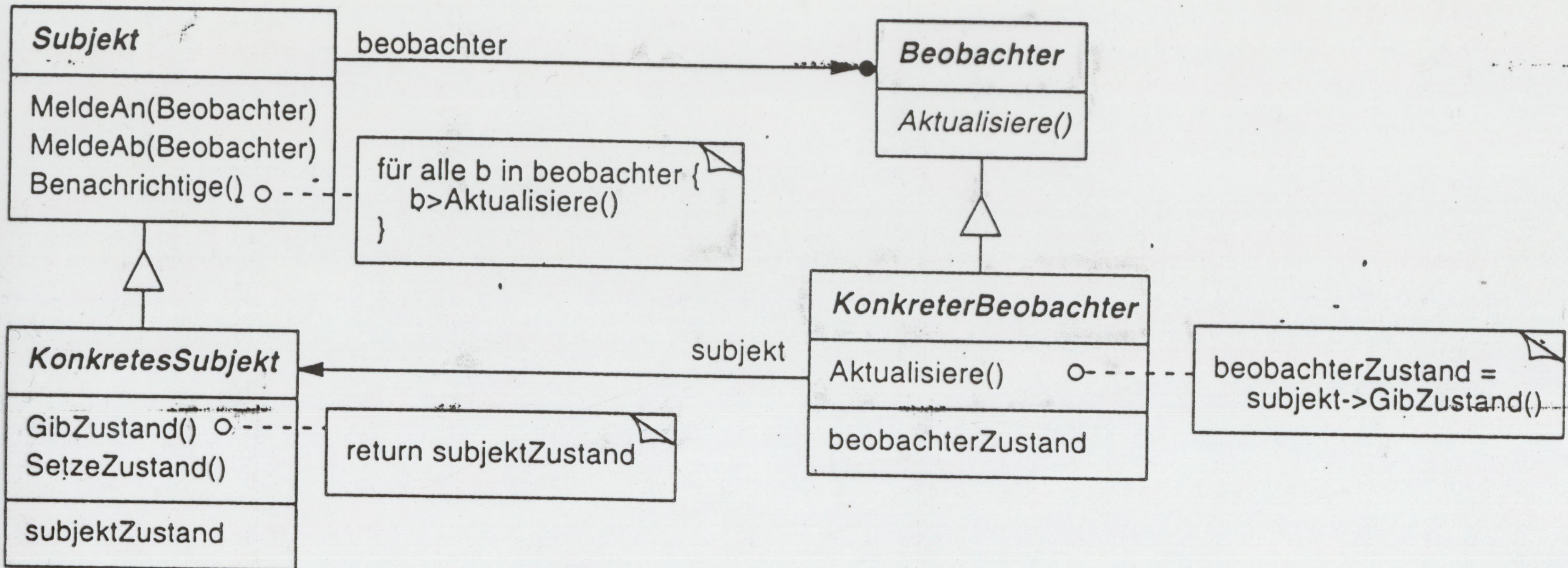
Beobachter (*Observer*)

Verhaltensmuster
objektbasiert

Definiere eine 1-zu-n-Abhängigkeit zwischen Objekten, so dass die Änderung des Zustands eines Objekts dazu führt, dass alle abhängigen Objekte benachrichtigt und automatisch aktualisiert werden.



Beobachter (*Observer*)



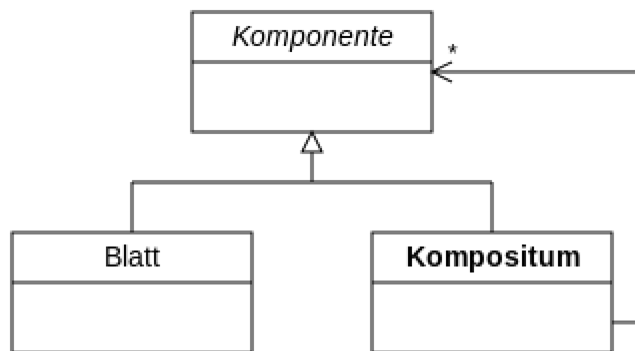
Beobachter (*Observer*)

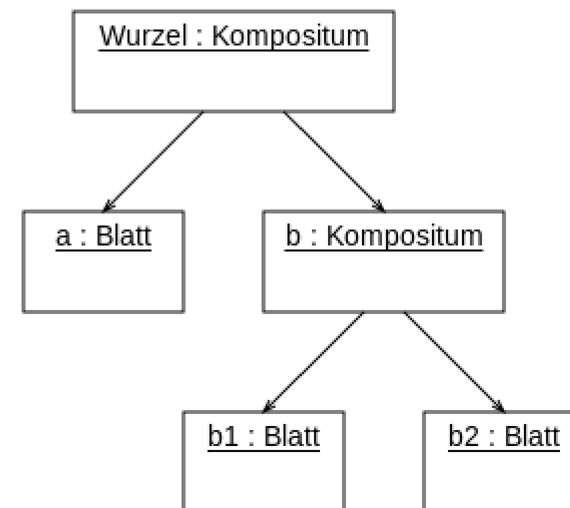
- Beispiel: Digitaluhren beobachten Zeitgeber
- minimale Kopplung bringt Vorteile
  - Beobachter und Subjekte können unabhängig voneinander modifiziert werden.
  - Subjekte und Beobachter können einzeln wiederverwendet werden.
  - Subjekt kennt nur den abstrakten Beobachter.
    => Neue Beobachter können ohne Änderung des Subjekts hinzugefügt werden.
- Nachteile
  - Optimierungen (z. B. Anzeigeperformance) kaum möglich, da generisch
  - immer alle Aktualisierungen, obwohl manche vielleicht nicht nötig wären

Kompositum (*Composite*)

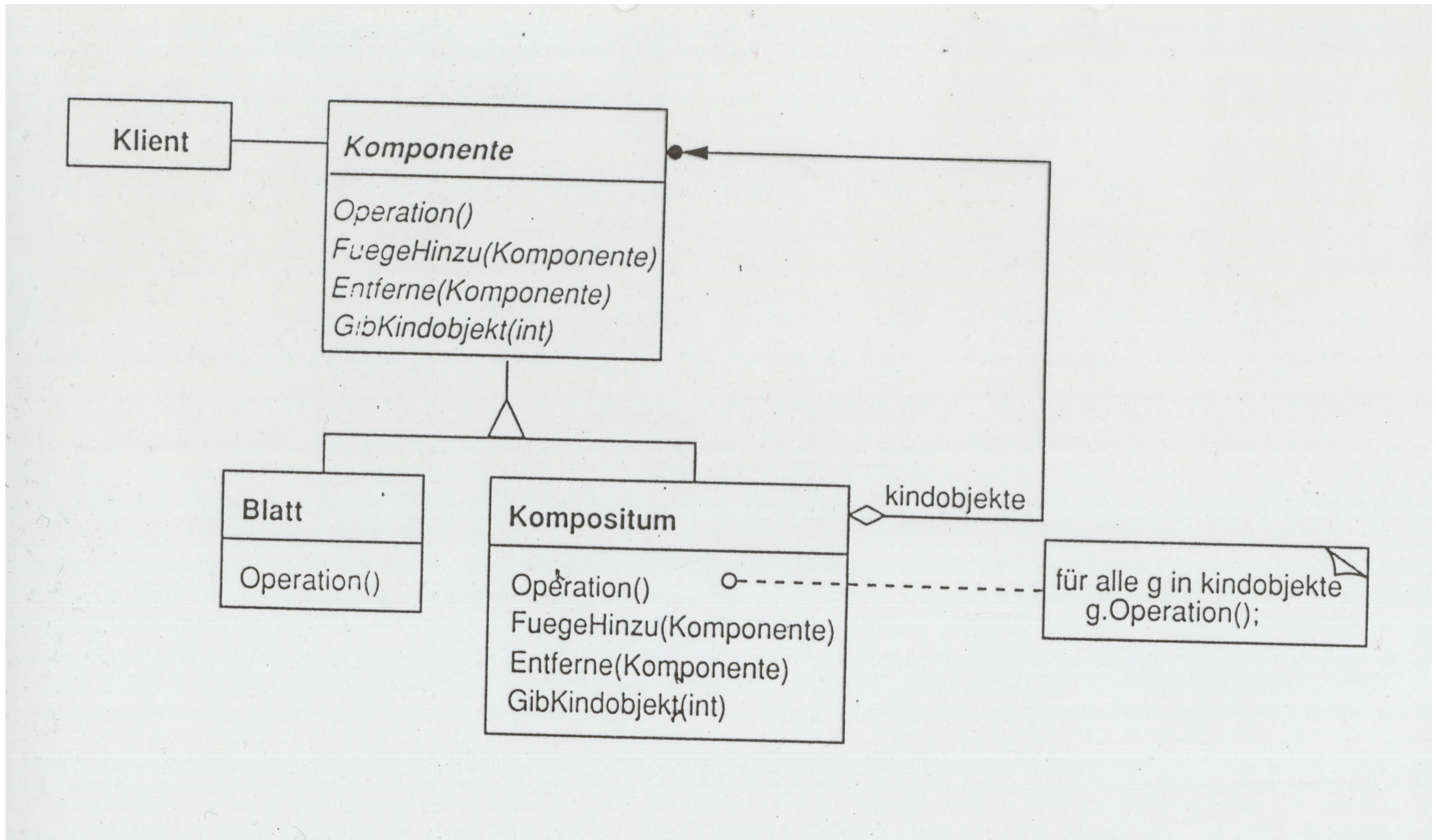
Strukturmuster
objektbasiert

- Füge Objekte zu Baumstrukturen zusammen, um Teile-Ganzes-Hierarchien zu modellieren. Das Muster ermöglicht es Klienten, einzelne Objekte sowie Kompositionen von Objekten einheitlich zu behandeln.
- Implementierung
  - explizite Referenzen auf Elternobjekte: erlaubt Aufwärtsnavigation
  - Blatt- und Kompositionsklassen vor Klienten versteckt





Kompositum (*Composite*)

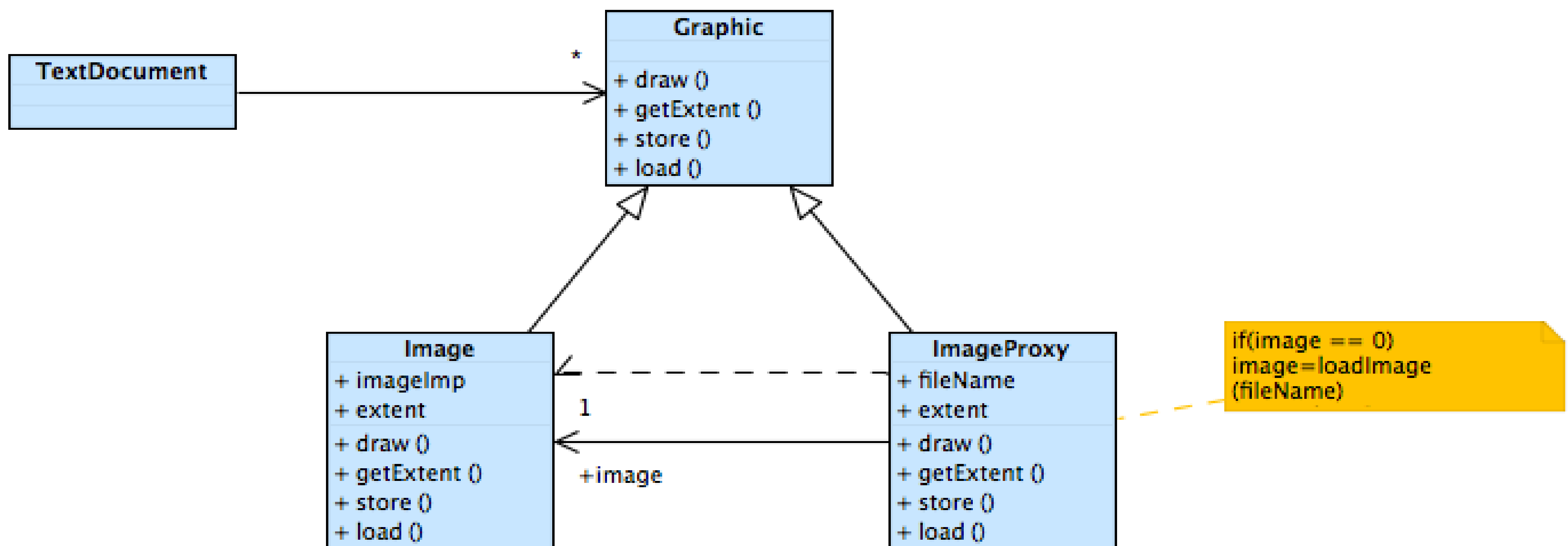


Kompositum (*Composite*)

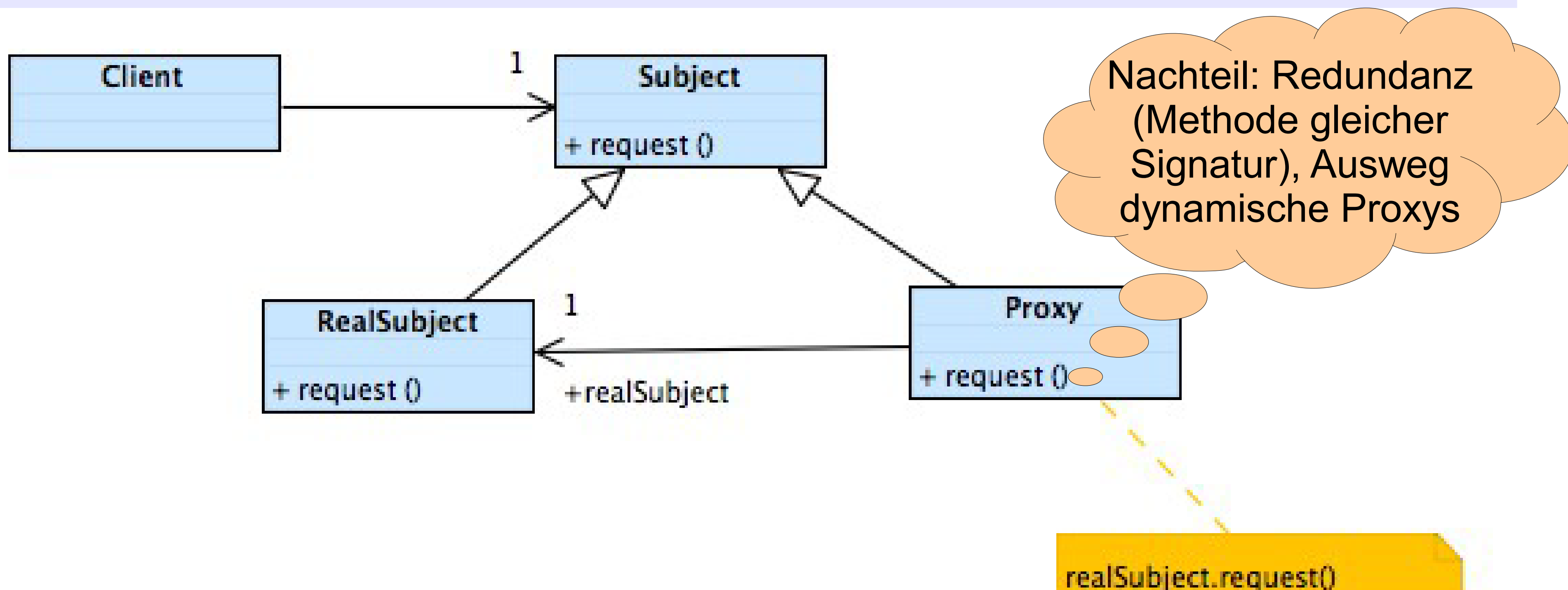
- Beispiele
  - hierarchische Dateisysteme
  - grafische Oberflächen und Malprogramme
- Klienten verwenden nur Schnittstelle von *Component*
- Vorteile
  - einheitliche Behandlung elementarer und zusammengesetzter Objekte
  - leichte Erweiterbarkeit um neue Blatt- und Containerklassen
- Nachteile
  - bei zu allgemeinem Entwurf Beschränkung auf bestimmte Klassen oder Typen erschwert
    => (unschöne) Typüberprüfungen zur Laufzeit nötig

Stellvertreter (*Proxy*)

Kontrolliere den Zugriff auf ein Objekt mit Hilfe eines vor-gelagerten Stellvertreter-Objekts.



Stellvertreter (*Proxy*)

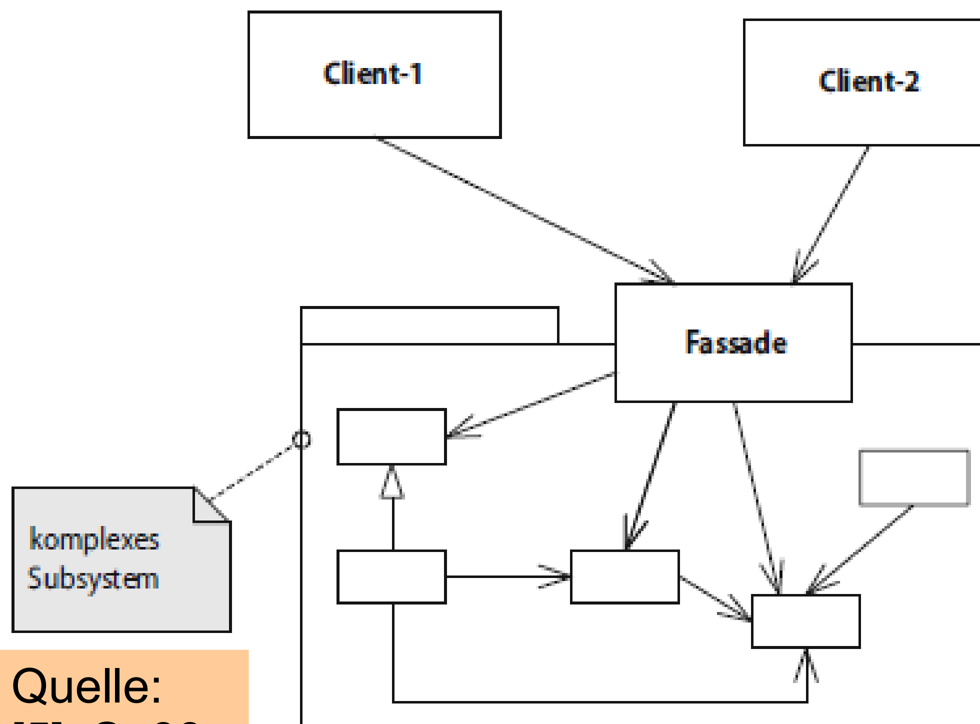


- Ein Proxy leitet Befehle an das echte Objekt weiter.
- Ein *Remote*-Proxy verbirgt die Tatsache, dass sich ein Objekt in einem anderen Adressraum befindet.
- Ein virtuelles Proxy dient der Optimierung, z. B. DB-Zugriff.
- Schutz-Proxys ermöglichen die Durchführung zusätzlicher Verwaltungsaufgaben beim Zugriff auf das Objekt.

Fassade (*Façade*)

Strukturmuster
objektbasiert

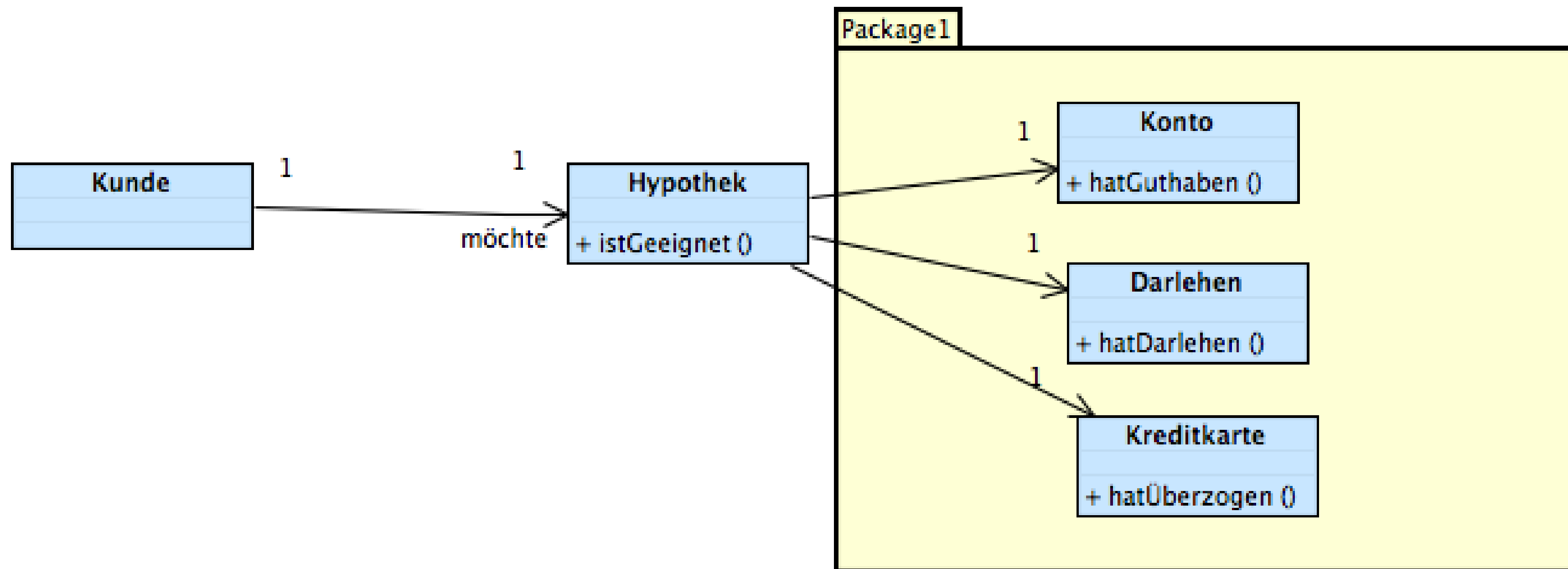
Biete eine einheitliche Schnittstelle zu einer Menge von Schnittstellen eines Subsystems. Die Fassadenklasse definiert eine abstrakte Schnittstelle, welche die Verwendung des Subsystems vereinfacht.



Quelle: [5], S. 88

Die Façade weiß, welche Klassen des Pakets für die Bearbeitung einer Botschaft zuständig sind und delegiert Botschaften an die zuständige Klasse.
Keine neue Funktionalität!

Fassade (*Façade*)

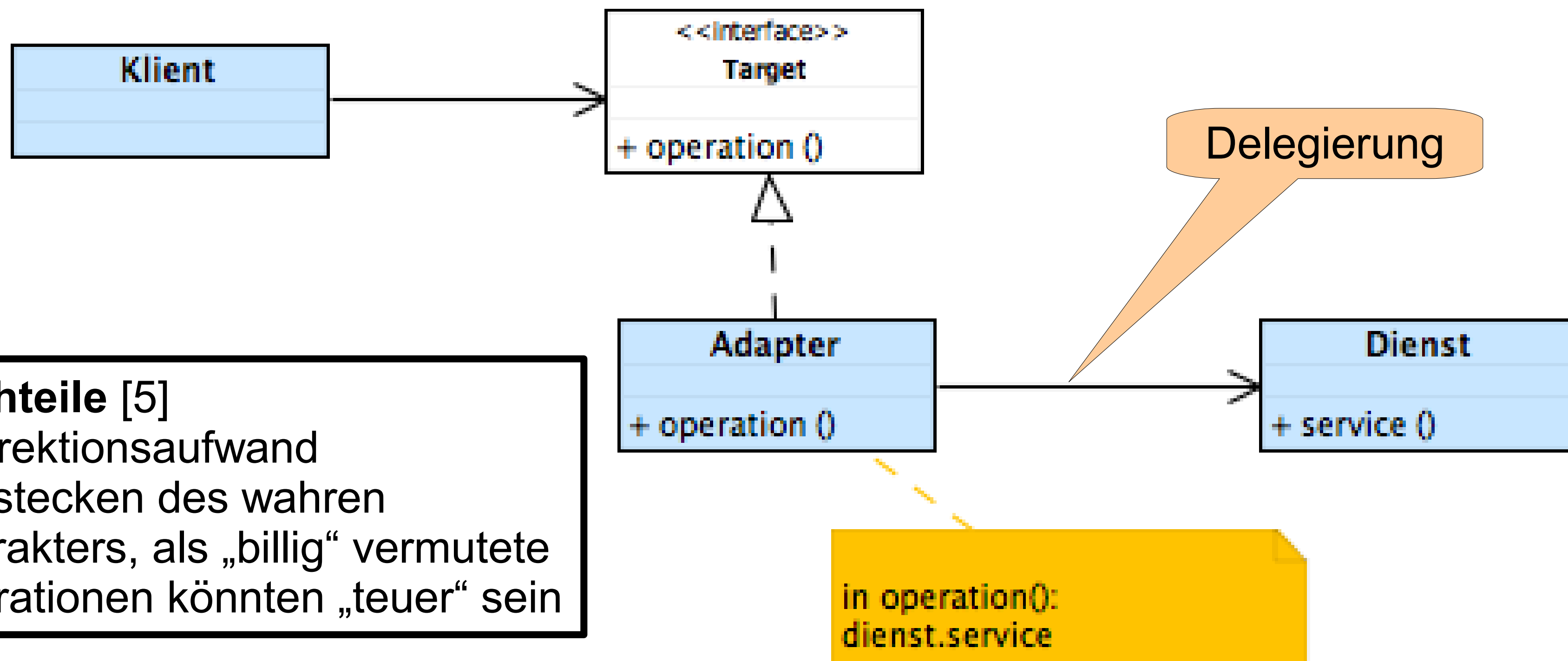


- Klienten kommunizieren mit dem Paket, indem sie Botschaften an die Fassade schicken, welche diese dann an das zuständige Objekt innerhalb des Pakets weiterleitet
- weniger Klassen, welche Klienten kennen müssen
- lose Kopplung erleichtert es, Pakete auszutauschen und deren unabhängige Implementierung
- bei Bedarf Umgehung, direkte Kommunikation mit Klassen
  - nur organisatorisch, jedoch nicht technisch zu verhindern [5]

Adapter (*Adapter, Wrapper*)

Strukturmuster
objektbasiert

Passe die Schnittstelle einer Klasse an eine andere, von Kunden erwartete Schnittstelle an. Zur Anpassung der Schnittstelle wird ein Adapterobjekt verwendet.



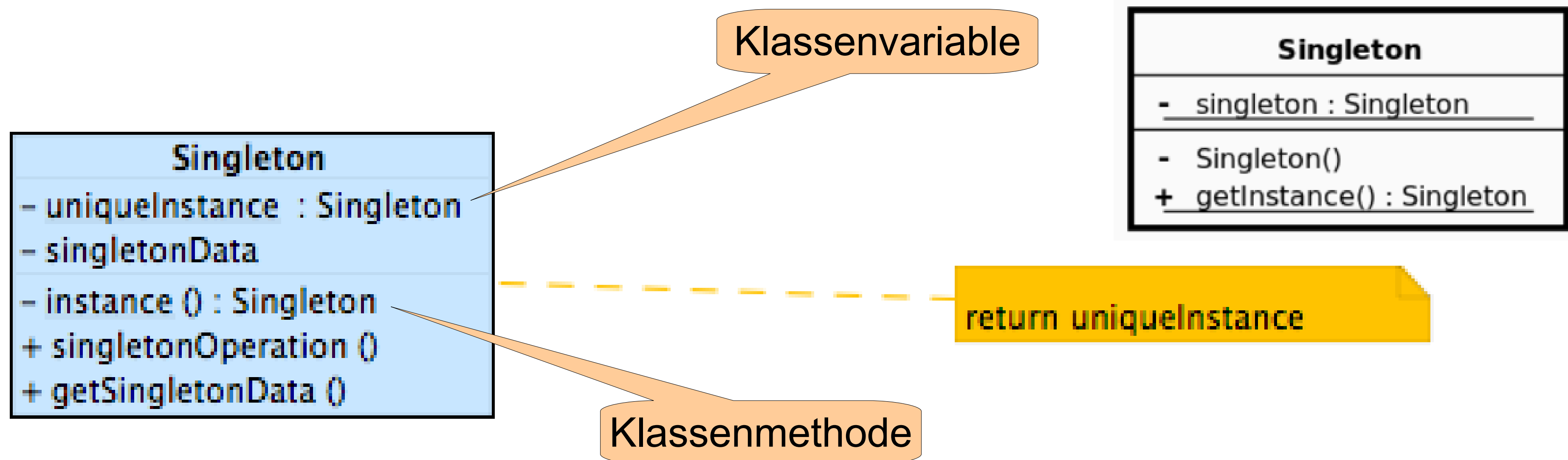
Nachteile [5]
-Indirektionsaufwand
-Verstecken des wahren Charakters, als „billig“ vermutete Operationen könnten „teuer“ sein

Beispiel: „Hüllenklassen“: Klasse `Integer` für Datentyp `int` usw. in *Java*

Einzelstück (*Singleton*)

Sichere ab, dass eine Klasse genau ein Exemplar besitzt, und stelle einen globalen Zugriffspunkt darauf bereit.

- bei manchen Klassen notwendig, dass es genau 1 Objekt gibt
- Zugriff von mehreren anderen Objekten erforderlich



Einzelstück (*Singleton*)

```
public final class Singleton {
    private static Singleton uniqueInstance = null;
    private Singleton(){
    }
    public static Singleton instance(){
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }
}
```

- Probleme bei *Multithread*-Anwendungen
  - Aufwand, Performance
  - Zerstörung (Referenzen?)

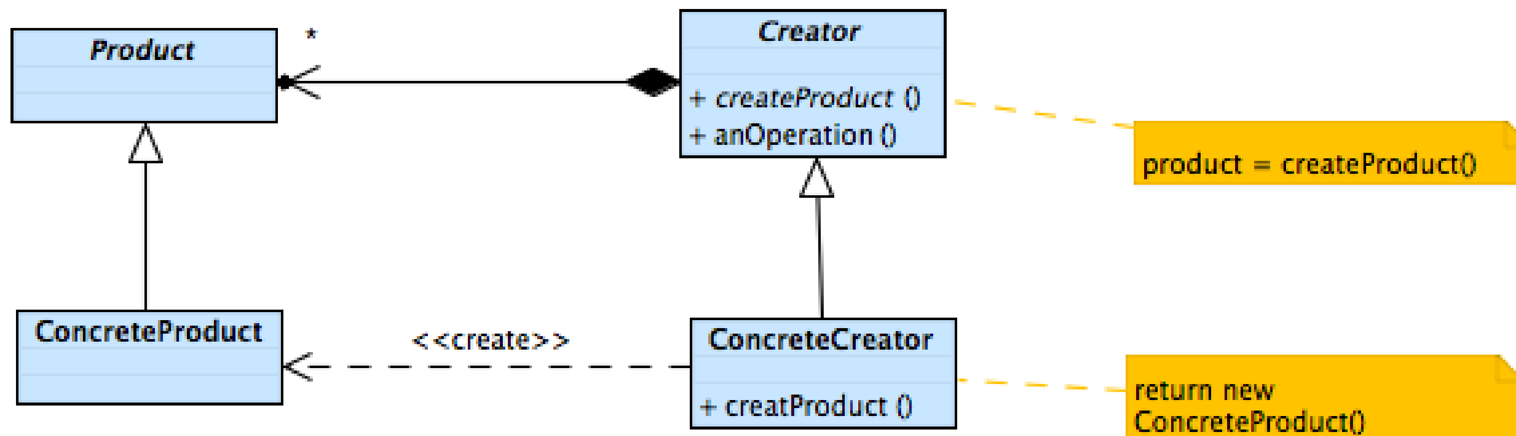
Quelle: [5]

- Verbesserung gegenüber globalen Variablen
  - Zugriffskontrolle: Konstruktor ist `private`, nur mittels `instance()` können Objekte erzeugt werden.
  - *Singleton*-Klasse kann durch Unterklassen spezialisiert werden.
  - falls später mehrere nötig, leichter änderbar als bei globaler Var.
- Vorsicht vor „Singletonitis“
  - Gefahr, prozedural statt objektorientiert zu programmieren

Fabrikmethode (*Factory Method*)

Erzeugungsmuster
klassenbasiert

- Definiere eine Klassenschnittstelle mit Operationen zum Erzeugen eines Objekts, aber lasse Unterklassen entscheiden, von welcher Klasse das zu erzeugende Objekt ist. Fabrikmethoden ermöglichen es einer Klasse, die Erzeugung von Objekten an Unterklassen zu delegieren.
- auch: „virtueller Konstruktor"

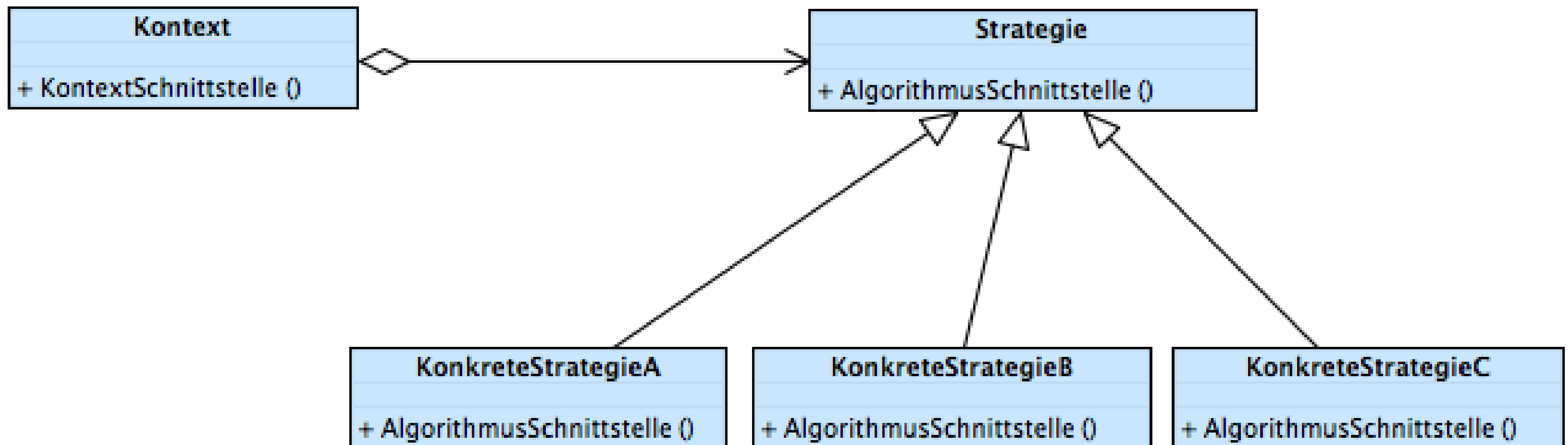


`createProduct()` ist abstrakt und heißt Fabrikmethode, sie ist für die Fabrikation von Objekten verantwortlich.

Strategie (*Strategy*)

Verhaltensmuster
objektbasiert

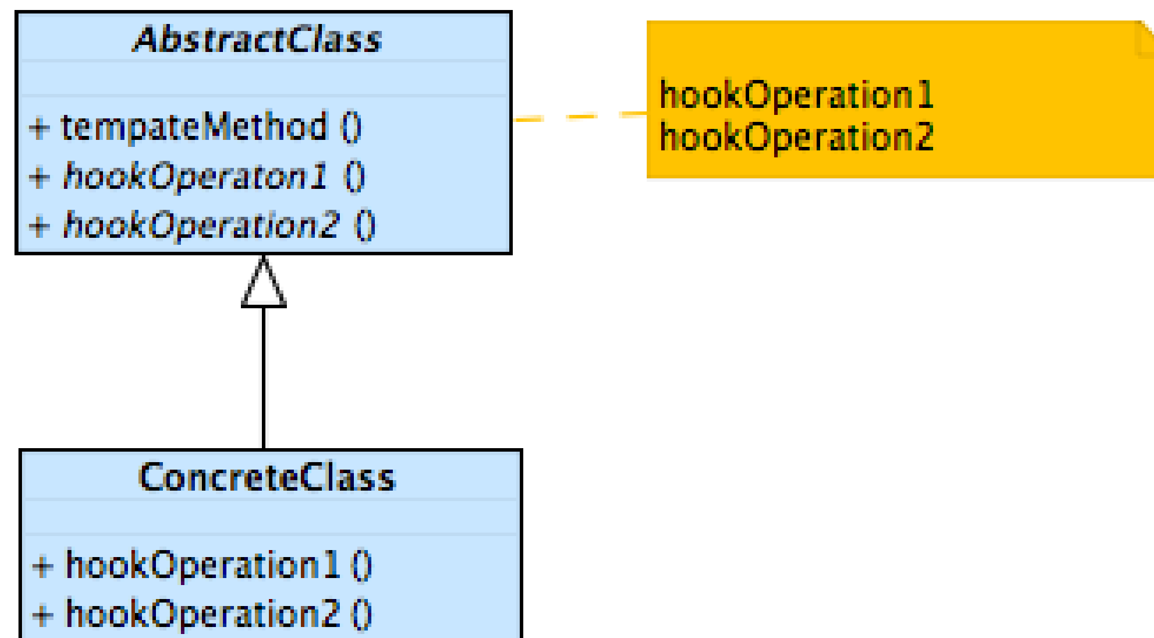
Definiere eine Familie von Algorithmen, kapsele jeden einzelnen und mache sie austauschbar. Das Strategie-Muster ermöglicht es, den Algorithmus unabhängig von ihn nutzenden Klienten zu variieren.



Schablonenmethode (*Template Method*)

Verhaltensmuster
objektbasiert

Definiere das Skelett eines Algorithmus in einer Operation und delegiere einzelne Schritte an Unterklassen. Die Verwendung einer Schablonenmethode ermöglicht es Unterklassen, bestimmte Schritte eines Algorithmus zu überschreiben, ohne seine Struktur zu verändern.



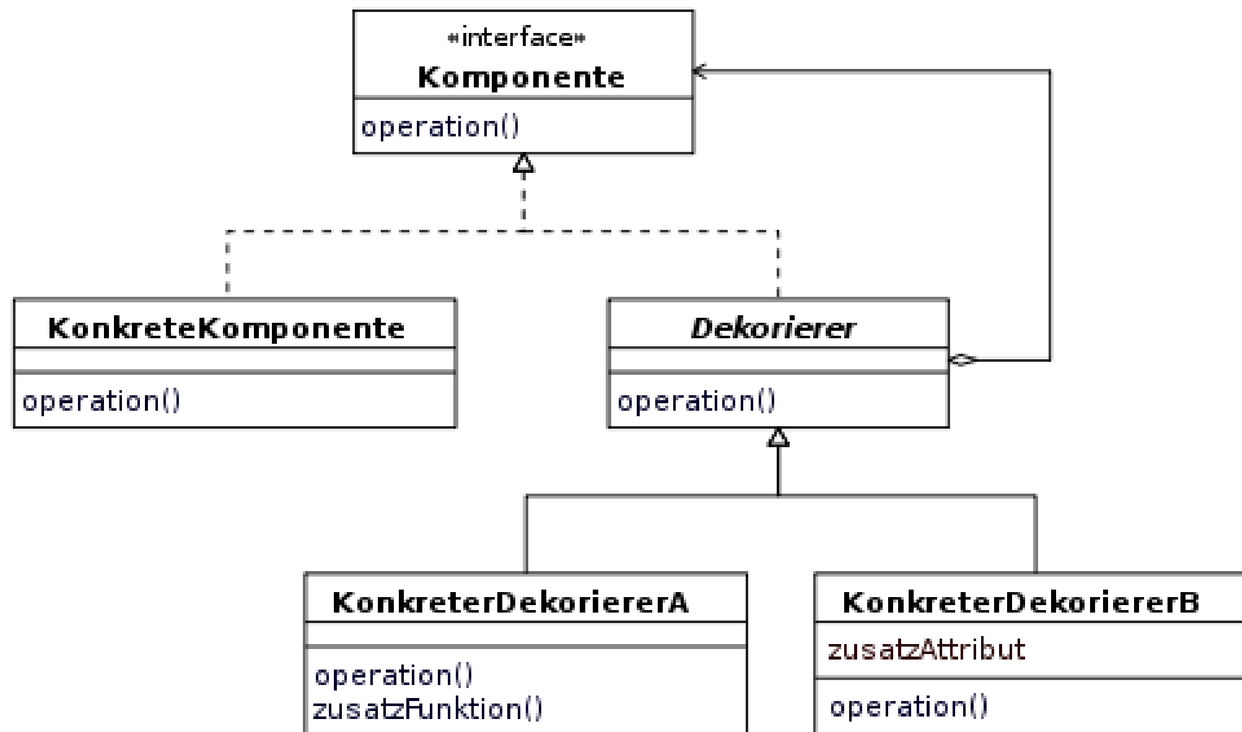
Schablonenmethode (*Template Method*)

- Schablonenmethoden bilden die grundlegende Technik zur Wiederverwendung von Code.
- Realisierung des Hollywood-Prinzips:
  „*Don't call us, we'll call you!*“

Dekorierer (*Decorator*)

Strukturmuster
objektbasiert

Erweitere ein Objekt dynamisch um Zuständigkeiten. Dekorierer bieten eine flexible Alternative zur Unterklassen-bildung, um die Funktionalität einer Klasse zu erweitern.



- Delegierung von Aufrufen an den Dekorierer
- Klient bekommt gar nicht mit, dass ein Dekorierer vorliegt.
- Vorteil: sehr flexibel, Austausch zur Laufzeit und sogar nach Instanziierung möglich
- **Nachteil**: intransparent, Objektgleichheit nicht mehr intuitiv

Nachteile

- Nutzung von Entwurfsmustern i. d. R. nicht zu Projektbeginn planbar
  - häufig erst bei Problemen während der Entwicklung Nachdenken über den Einsatz eines Musters
  - Auswahl eines passenden unter den 23 GoF-Mustern mit etwas Erfahrung möglich
  - aber: > 100 Entwurfsmuster vorgeschlagen, Spezialfall könnte trotzdem nicht abgedeckt sein
- viel Erfahrung nötig, um zu entscheiden, ob ein Entwurfsmuster passen könnte oder ob eine Speziallösung nötig ist

Zusammenfassung

- Entwurfsmuster als großartiger Ansatz
  - Schwachpunkt der fest vorgegebenen Algorithmen und Schnittstellen bei fertigen Komponenten angegangen
  - Wiederverwendung der Ideen und Ansätze anstelle einer Implementierung
- Entscheidung, ob ein Entwurfsmuster passt oder nicht, ist nicht leicht
  - Eine Speziallösung kann besser sein als ein krampfhaft hingebogenes Muster.
  - Künstliche Verwendung möglichst vieler Entwurfsmuster wird sogar als *Antipattern* angesehen.
  - Die Wiederverwendung grobgranularer Architekturmuster (z. B. MVC-Muster) scheint für nicht so erfahrene Entwickler günstiger.

Literatur

[1] Ian Sommerville: „Software Engineering 10“, Addison-Wesley, 2016

[2] Erich Gamma, Richard Helm, Ralph E. Johnson, John Vlissides: Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software. Addison Wesley, München 2004, ISBN 3-8273-2199-9 (Englisch: *Design Patterns. Elements of Reusable Object-Oriented Software*. 1994)

[3] Christopher Alexander, Sara Ishikawa, Murray Silverstein: „A Pattern Language: Towns, Buildings, Construction“, Oxford University Press 1977

[4] James Coplien: Advanced C++ Programming Styles and Idioms. Addison-Wesley 1991

[5] Karl Eilebrecht, Gernot Starke: „Patterns kompakt“, Springer, 5. aktualisierte und erweiterte Auflage, 2019