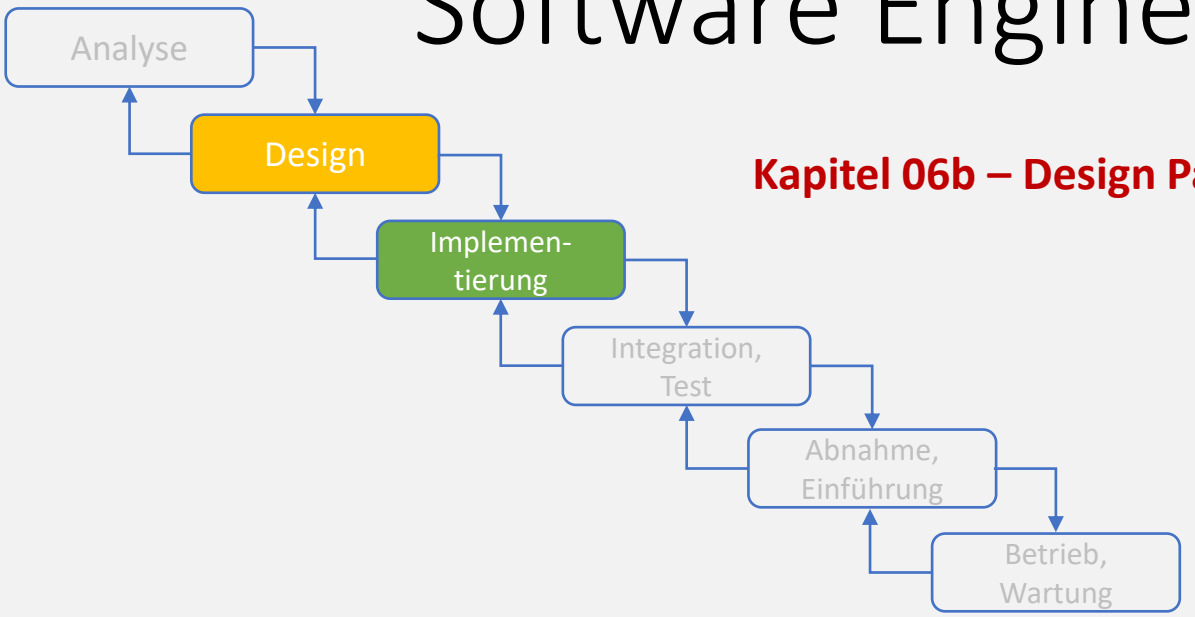
Software Engineering



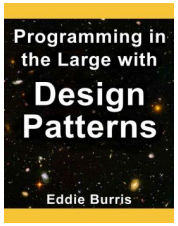
Kapitel 06b – Design Patterns

OSTBAYERISCHE TECHNISCHE HOCHSCHULE REGENSBURG
INFORMATIK UND MATHEMATIK

Gliederung

1. Motivation und Initialbeispiel
2. Geschichte der Design Patterns
3. Was sind Design Patterns?
4. Nutzung von Design Patterns
5. Template für Design Patterns
6. Liste wichtiger Design Patterns
7. Diskussion ausgewählter Design Patterns

Hinweis: die folgenden Folien basieren auf den Folien von Eddie Burris und seinem Buch: "Programming in the Large with Design Patterns"



Motivation und Initialbeispiel

[Bildquelle: http://www.philipphauer.de]

Motivation und Initialbeispiel

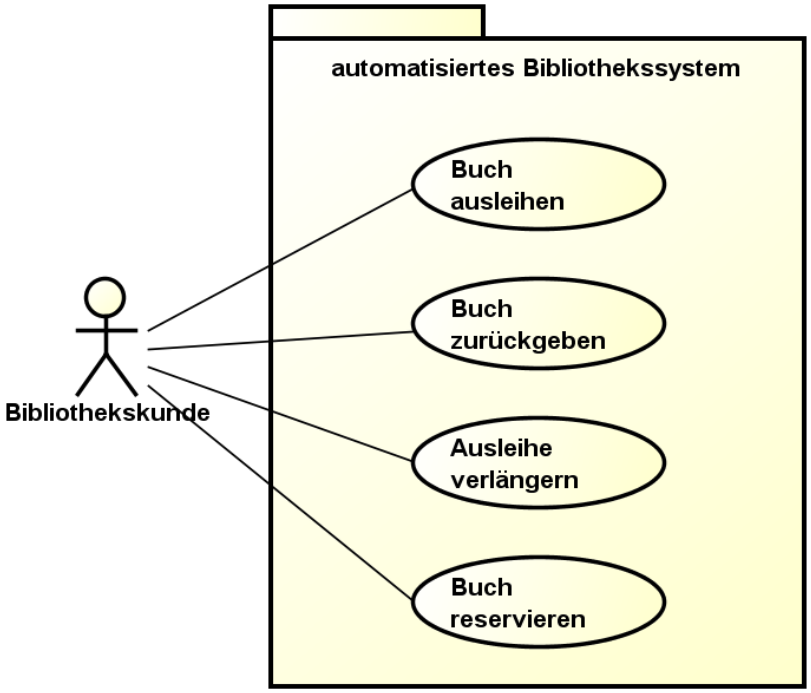
Angenommen, wir befinden uns in den letzten Zügen des Designs für ein automatisiertes Bibliothekssystem

- **Die Anforderungen lauten:** der Kunde kann…
  - Ein Buch ausleihen
  - Ein Buch zurückgeben
  - Die Ausleihe eines Buches verlängern
  - Ein Buch reservieren

Wie wäre jetzt das weitere Vorgehen?

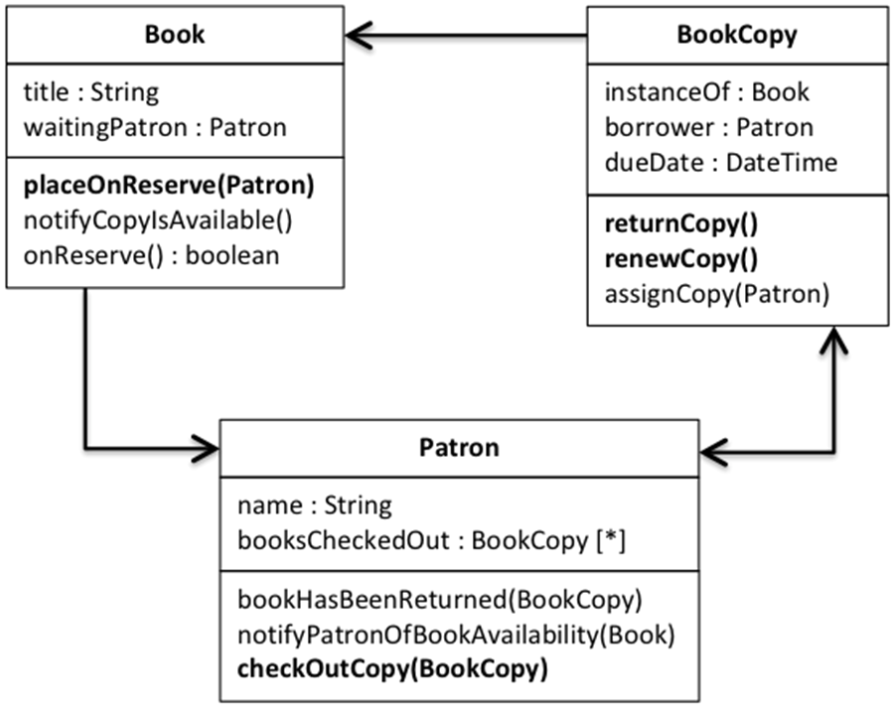
Motivation und Initialbeispiel

Use Case Diagramm für unser automatisiertes Bibliothekssystem:



Motivation und Initialbeispiel

Initialer Entwurf:

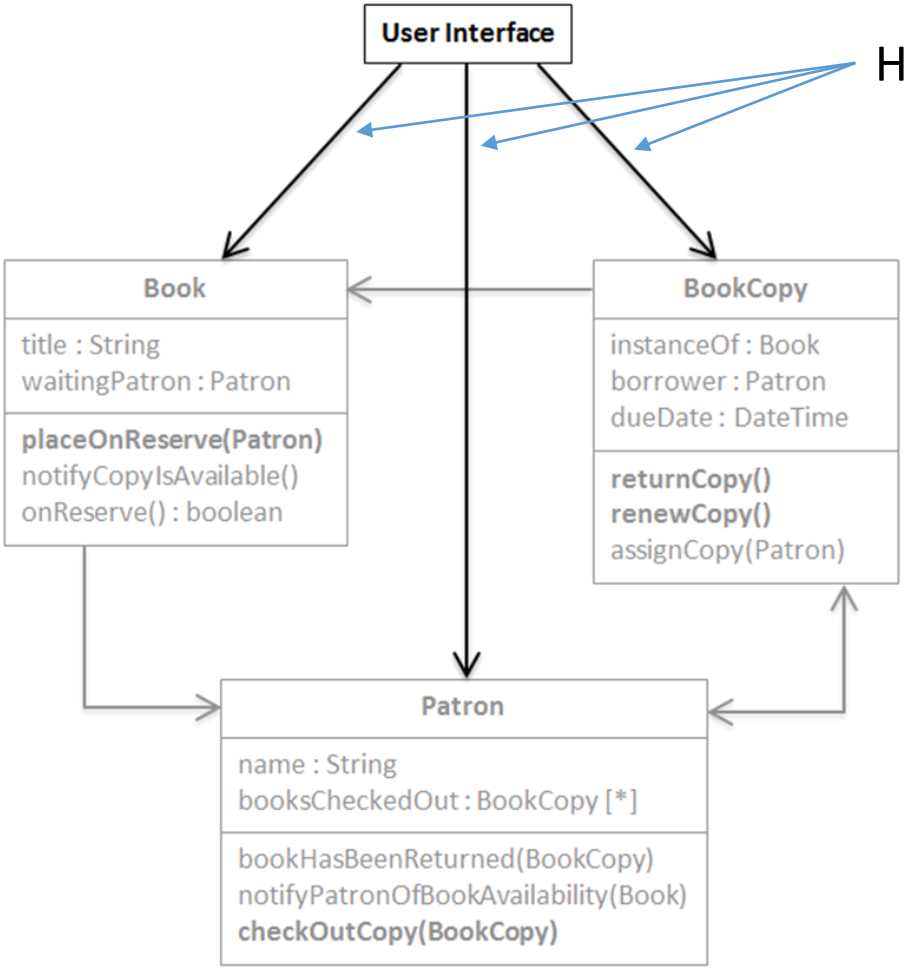


Welche Probleme ergeben sich?

- Wesentliche Funktionen (s. Use Cases) über alle Klassen verteilt
- Hohe Kopplung
- Z.B. grafisches User Interface ist von allen drei Klassen abhängig

Wie sähe das Klassendiagramm aus, wenn man eine GUI-Komponente integrieren würde?

Motivation und Initialbeispiel

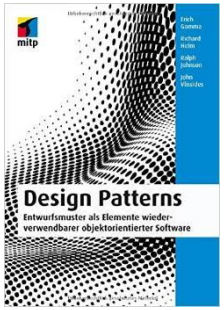


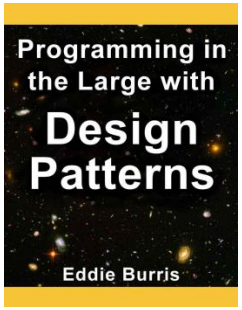
Hohe Kopplung!

Was tun?

Motivation und Initialbeispiel

- Suche nach Lösungen in Lehrbüchern:





- **Singleton** – Sichere ab, dass eine Klasse genau ein Exemplar besitzt und stelle einen globalen Zugriff darauf bereit.

Hier keine Lösung

Motivation und Initialbeispiel

- **Iterator** – Biete eine Möglichkeit, um auf die Elemente eines zusammengesetzten Objektes sequentiell zugreifen zu können…

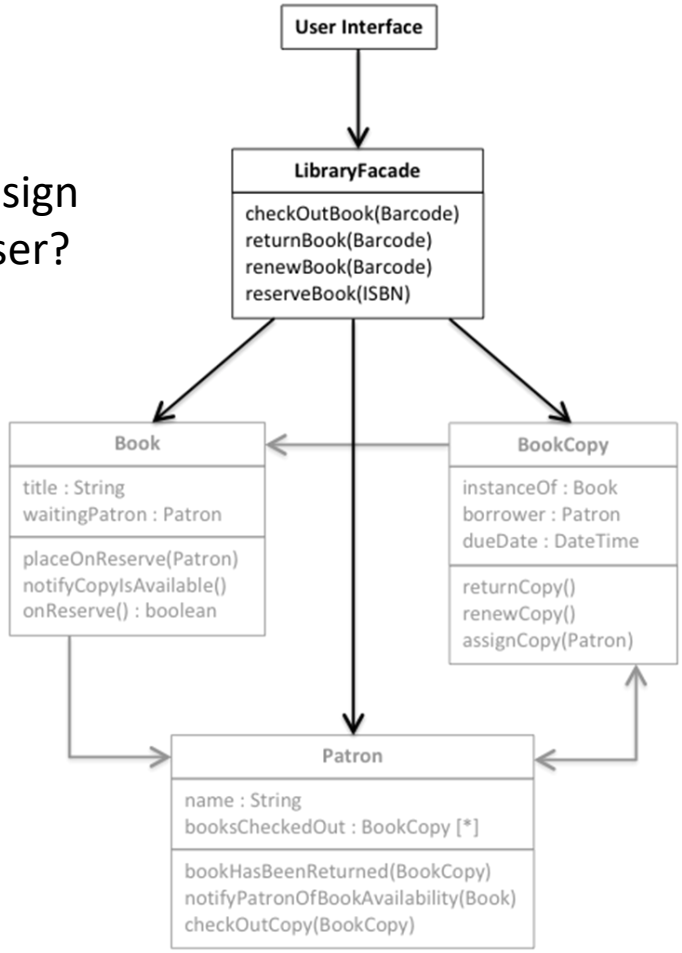
- …

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

Motivation und Initialbeispiel

Resultat nach Verwendung des Fassade-Musters:

Ist dieses Design wirklich besser?



Vorteile:

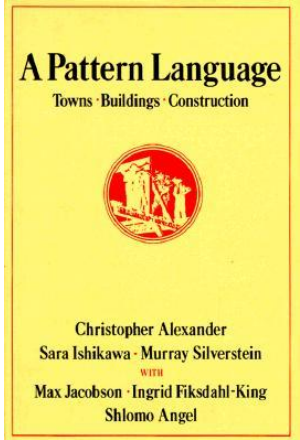
- geringere Kopplung,
  da User-Interface-Komponente i.a. aus vielen Klassen besteht

- Vorher:
  - Alle Klassen des UI hatten Referenzen zu den drei Klassen
  - Die Interfaces waren medium komplex
- Nachher:
  - Alle UI-Klassen haben nur noch eine Referenz
  - Sie greifen über ein einfaches Interface auf die drei Klassen zu

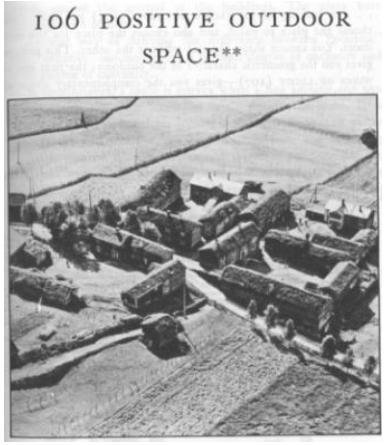
Geschichte der Design Patterns

Wo stammen sie her?

- Patterns wurden nicht im Zusammenhang mit Software sondern im Zusammenhang mit Städteplanung und Gebäudearchitektur erfunden
- Christopher Alexander (Architekturtheoretiker) erfand in den 70ern eine Pattern-Sprache, die half, bessere Gebäude zu bauen
- Das erste Buch „*Eine Muster-Sprache. Städte, Gebäude, Konstruktion*“ beinhaltete bereits 253 solche Muster

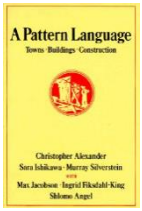


- Jedes Muster bot eine allgemeine Lösung bzw. ein bewährtes Verfahren zur Lösung typischer, wiederkehrender Probleme



Geschichte der Design Patterns

1977:
Christopher Alexander als Coauthor des ersten Buchs über Muster „*Eine Muster-Sprache. Städte, Gebäude, Konstruktion*“.
Es dokumentiert 253 Muster im Bereich Städteplanung und Gebäudearchitektur.



1987:
Kent Beck und Ward Cunningham schlagen vor, eine Muster-Sprache analog zu der von Christopher Alexander für Software Design einzuführen

1994:
Die “Gang of Four” starten die Software-Muster-Bewegung mit ihrer Publikation des ersten Buchs über Software Patterns.
In ihm werden 23 mid-level Software-Design-Muster beschrieben.

Was sind Design Patterns

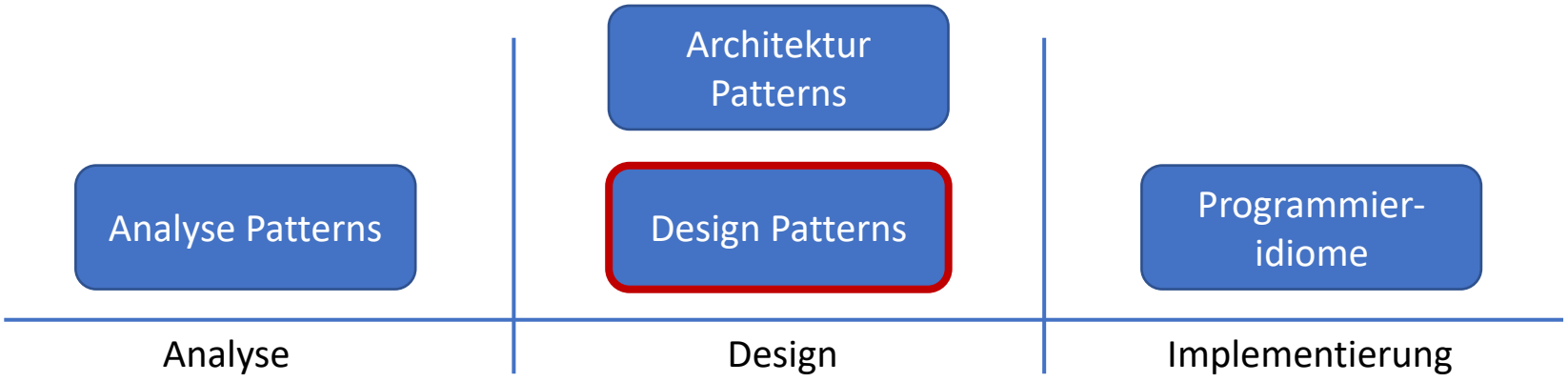
Definition:

- Design Patterns sind wiederverwendbare Lösungen zu wiederkehrenden Design-Problemen

Was sind Design Patterns nicht?

- Sie sind kein konkretes Design
- Sie sind keine konkrete Implementierung (wie etwa ein Algorithmus)

Abgrenzung:



Was sind Design Patterns

Design Patterns liefern eine Transformation von einem spezifischen Design-Problem in eine generische Lösung

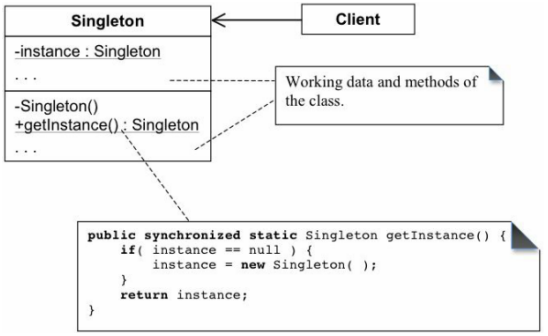
Design-Problem:

Wie kann ich sicherstellen, dass eine Klasse nur eine Instanz besitzt?

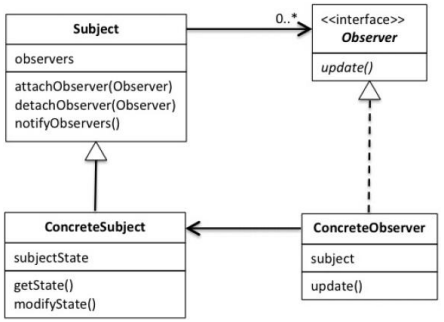
Wie kann ich abhängige Objekte über Zustandsänderungen informieren, ohne dass sie dauernd anfragen müssen?

Generische Lösung:

Singleton

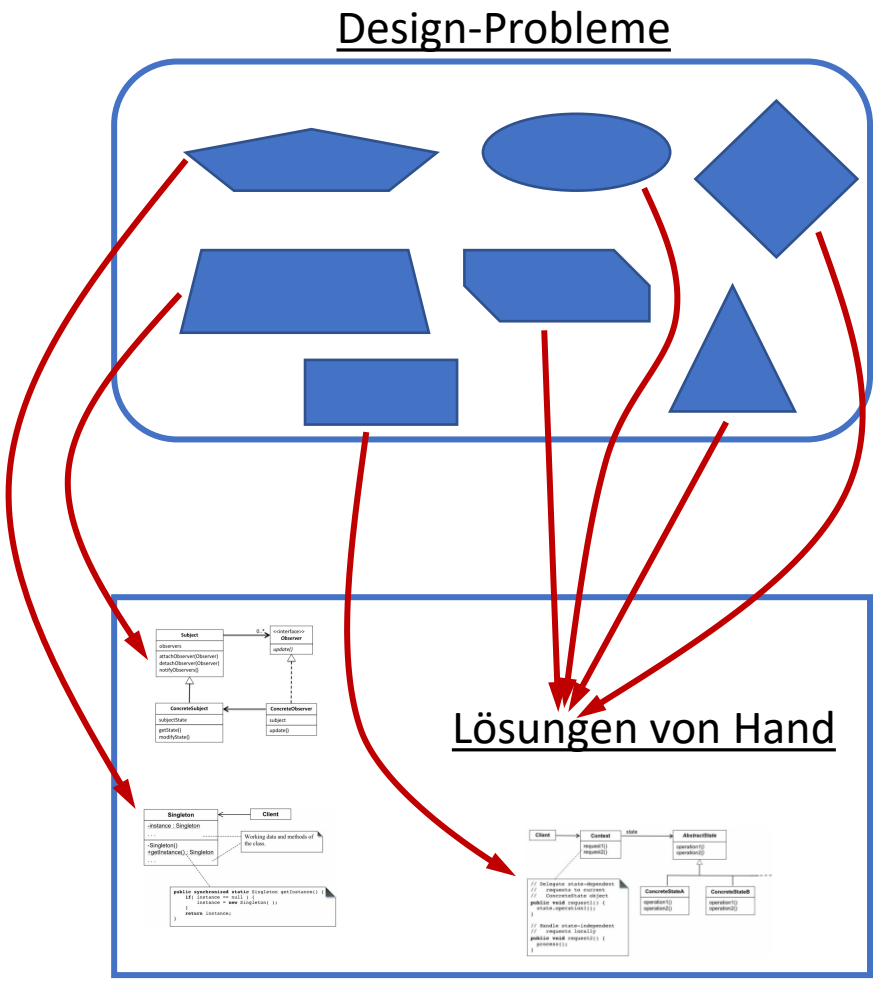


Observer



Was sind Design Patterns

- Kenntnis von Design Patterns vereinfacht Software Design durch Reduzierung der Anzahl von Design Problemen die von Hand gelöst werden müssen
- Design Probleme, die auf bereits dokumentierten Design Patterns passen, besitzen fertige Lösungen
- Design Probleme, die keinem existierenden Pattern entsprechen, müssen von Hand gelöst werden



Nutzen von Design Patterns

Design Patterns:

- Unterstützen Wiederverwendung,
- Vereinfachen das Design,
- Definieren eine gemeinsame Sprache,
- Erleichtern Dokumentation und Wissenstransfer,
- Erleichtern ingenieurmäßiges Vorgehen bei der Softwareentwicklung,
- Fördern Fähigkeit, selbständig gutes Design zu erstellen,
- Erleichtern den Zugang zu Klassenbibliotheken und Frameworks,
- Erleichtern den Zugang zu objektorientierten Sprachen

Design-Pattern-Template

Inhalt	Beschreibung
Patternname	Kurzer beschreibender Name
Motivation	Motivation das Pattern zu lernen
Zweck	Beschreibung des Design Problems, das das Pattern lösen soll
Lösung	Strukturdiagramm, Interaktionsdiagramm (grafische Repräsentation des Patterns)
Beispielcode	Code-Fragment, das eine Beispielimplementierung des Patterns zeigt
Diskussion	Diskussion; Probleme, die sich bei der Implementierung oder Nutzung des beschriebenen Pattern ergeben können
Verwandte Patterns	Patterns im Zusammenhang mit dem beschriebenen Pattern

Liste wichtiger Patterns

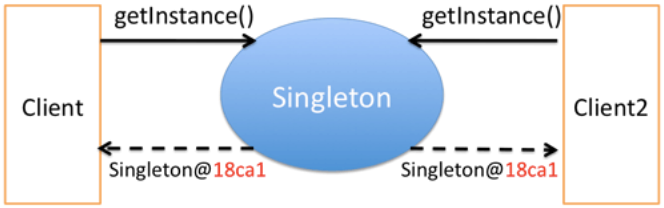
Erzeugermuster
- Singleton
- Factory Method
- Abstract Factory
- …

Strukturmuster
- Adapter
- Decorator
- Façade
- …

Verhaltensmuster
- Iterator
- State
- Strategy
- Observer
- Template Method
- …

Diskussion ausgewählter Design Patterns

Singleton: Ziel und Idee (Einordnung: Erzeugermuster)

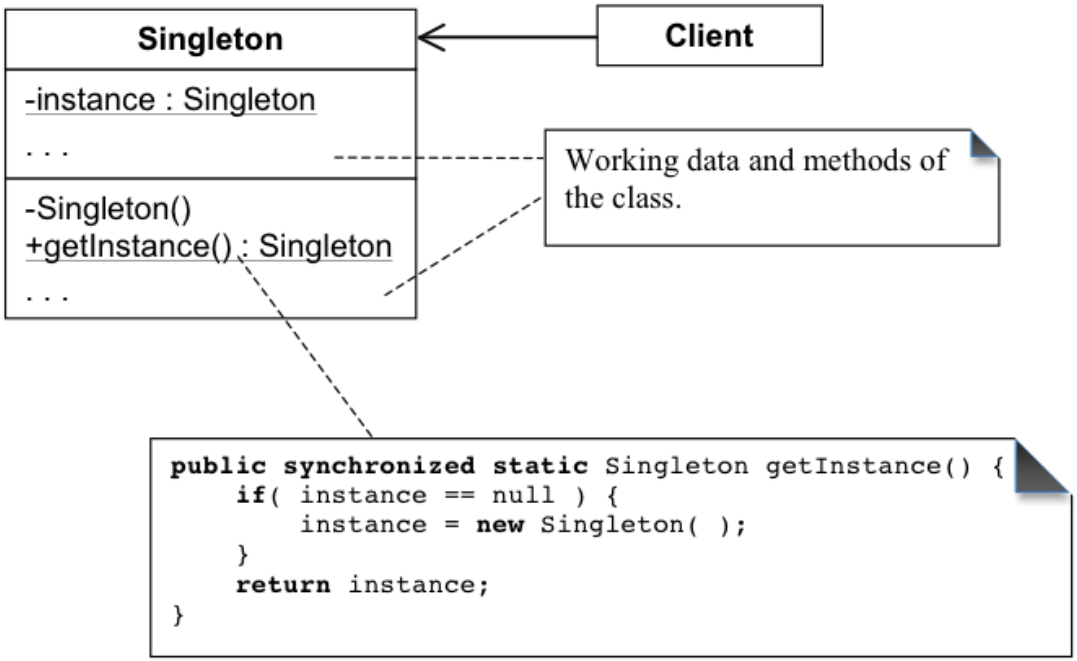


[Bild: www.philipphauer.de]

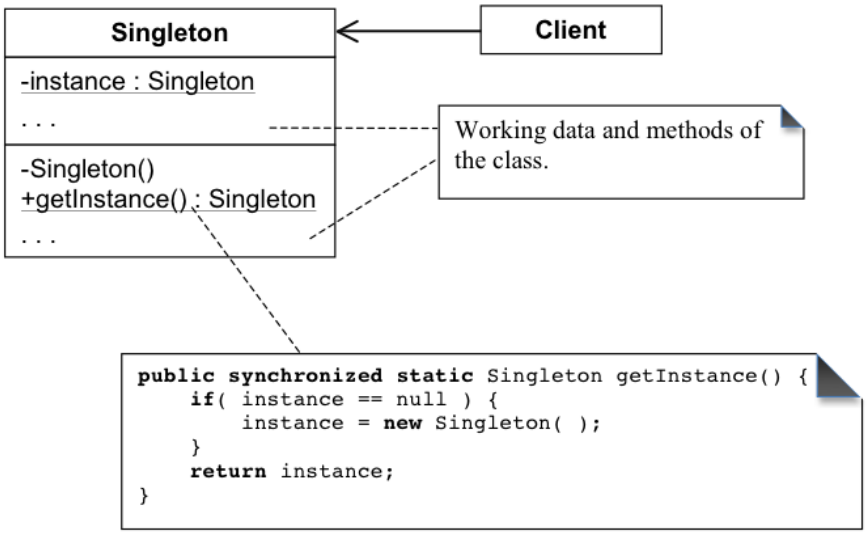
- **Ziel: das Singleton-Design-Pattern…**
  - Stellt sicher, dass nicht mehr als eine Instanz einer Klasse erzeugt werden kann
  - Stellt einen globalen Zugriffspunkt zu dieser Instanz bereit

- **Lösungsidee:**
  - Konstruktor privat machen, um zu verhindern, dass Clients Instanzen erzeugen können
  - Öffentliche Methode `getInstance()` erstellen, die einzige Instanz der Klasse zurückgibt
  - Beim ersten Aufruf wird die Instanz erzeugt, gecached und zurückgegeben
  - Bei nachfolgenden Aufrufen wird nur die gecachte Instanz zurückgegeben

Singleton: Lösung



Singleton: Diskussion



- **Performance:**
  - `synchronized`: macht `getInstance()` thread-safe
  - es kann aber die Performance beeinträchtigen
- **falls Thread-safety kein Thema ist:**
  - auf Schlüsselwort `synchronized` verzichten

Andere Möglichkeiten?

Singleton: weiterer Ansatz

Alternative:

- Falls Aufwand eine Instanz zu erzeugen klein ist, oder
- Falls die Instanz auf jeden Fall erzeugt werden soll:

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

    public static Singleton getInstance() {
        return instance;
    }
}
```

- **Anwendbar, wenn:**
  - Alle zur Initialisierung notwendigen Daten bei der statischen Initialisierung zur Verfügung stehen

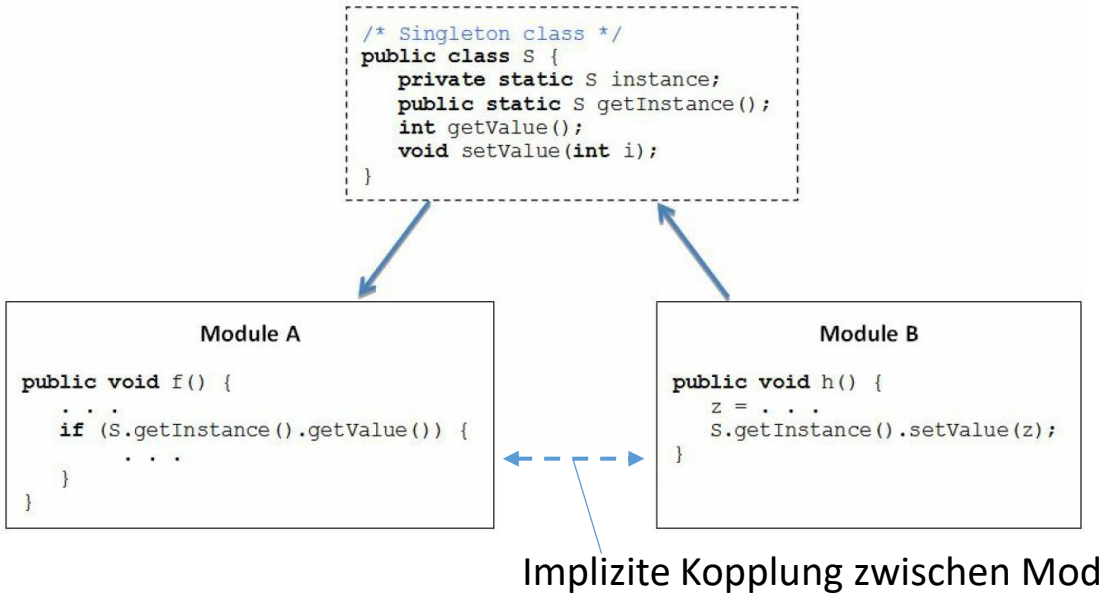
Singleton: Beispielanwendung

```
public class SimpleLog {

    private static SimpleLog instance = null;

    private SimpleLog() {
        // Open log files
        // ...
    }

    public synchronized static SimpleLog getInstance() {
        if (instance == null) {
            instance = new SimpleLog();
        }
        return instance;
    }

    // working method of class
    public synchronized void debug (String message) {
        // ...
    }

    public synchronized void info (String message) {
        // ...
    }

}
```

- Clientzugriff:

```
public class Client {

    public void someFunction() {
        SimpleLog logging = SimpleLog.getInstance();

        logging.debug("Client started function: Client::someFunction");

        // ...
    }

}
```

Singleton: Diskussion



- **Problem:** implizite Kopplung zwischen den Modulen wird durch geteilten Zugriff auf das Singleton verursacht

Iterator: Ziele und Ideen (Einordnung: Verhaltensmuster)

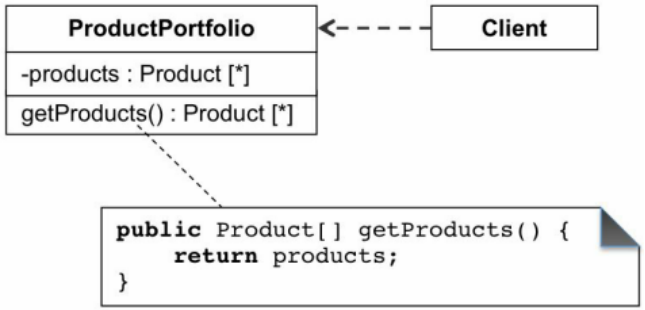
- **Ziel: das Iterator-Design-Pattern…**
  - Bietet einen einheitlichen Weg zur Traversierung aller Elemente eines Collection-Objekts
- **Beispiele für Collection-Objekte:**
  - Listen, Sets, Dictionaries, …
  - konkret: Produktportfolio, das alle Produkte eines Händlers speichert:

```
public class ProductPortfolio {

    private Product products[];

    // ...

}
```

- **Problemstellung:**
  - Klienten einer Collection müssen oftmals sequentiell auf jedes Element der Collection zugreifen

Iterator: 1. (naive) Möglichkeit

- **Idee:**
  - Erlaube dem Klienten den Zugriff auf die interne Datenstruktur
- **Ergebnis:**



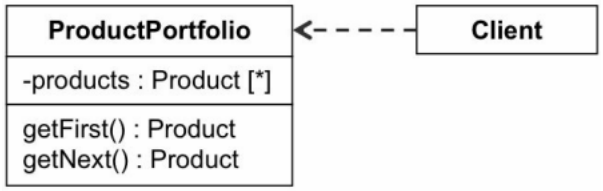
- **Probleme:**
  - Information-Hiding-Prinzip verletzt (Client kann direkt löschen und hinzufügen)
  - Starke Kopplung (durch Zugriff des Klienten auf interne Datenstruktur)
  - Änderungen an interner Struktur haben Auswirkungen auf Client

Iterator

- **Ziel: das Iterator-Pattern löst das obige Problem, so dass…**
  - der Client-Code lose mit dem Collection-Objekt und
  - dem zugrundeliegenden Iterations-Algorithmus gekoppelt ist

- Wir versuchen jetzt eine Lösung für das obige Ziel herzuleiten

Iterator: 1. Ansatz




```java
public class ProductPortfolio {

    private Product products[];

    public Product getFirst() {
        // returns first element
        // ...
    }

    public Product getNext() {
        // returns next element (if exists)
        // ...
    }

}
```

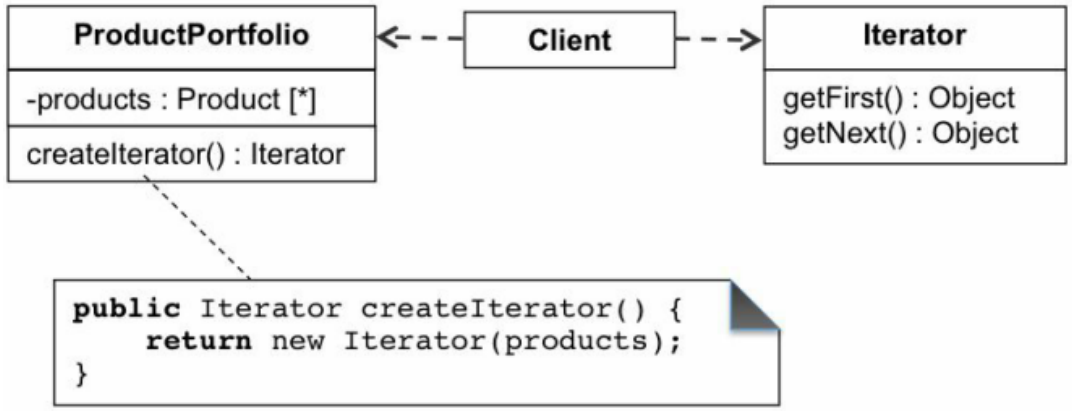
```java
public static void clientCode(ProductPortfolio portfolio) {
    Product product = portfolio.getFirst();

    while (product != null) {
        process(product);
        product = portfolio.getNext();
    }
}
```

- **Besser, aber:**
  - Verletzung des Single-Responsibility-Prinzips, denn:

    Portfolio Klasse ist nicht nur für Operationen zuständig, die für das Portfolio relevant sind, sondern auch für die Iteration

Iterator: 2. Ansatz (Verbesserung)

- **Verbesserung:**
  - Ziehe die Iterations-Logik aus der Portfolioklasse heraus
  - Implementiere Iterations-Logik in eigener Klasse `Iterator`



Iterator: 2. Ansatz (Verbesserung)

- **Code:**

```
public class ProductPortfolio {

    private Product products[];

    public Iterator createIterator() {
        return new Iterator (products);
    }

}
```

```
public class Iterator {

    private Product products[];

    public Iterator (Product products[]) {
        this.products = products;
    }

    public Product getFirst() {
        // return first element
        // ...


    }

    public Product getNext() {
        // return next element (if exists)
        // ...


    }

}
```

```
public static void clientCode(ProductPortfolio portfolio) {
    Iterator iterator = portfolio.createIterator();
    Product product = iterator.getFirst();

    while (product != null) {
        process(product);
        product = iterator.getNext();
    }
}
```

Iterator: 2. Ansatz (Verbesserung)

Diskussion:

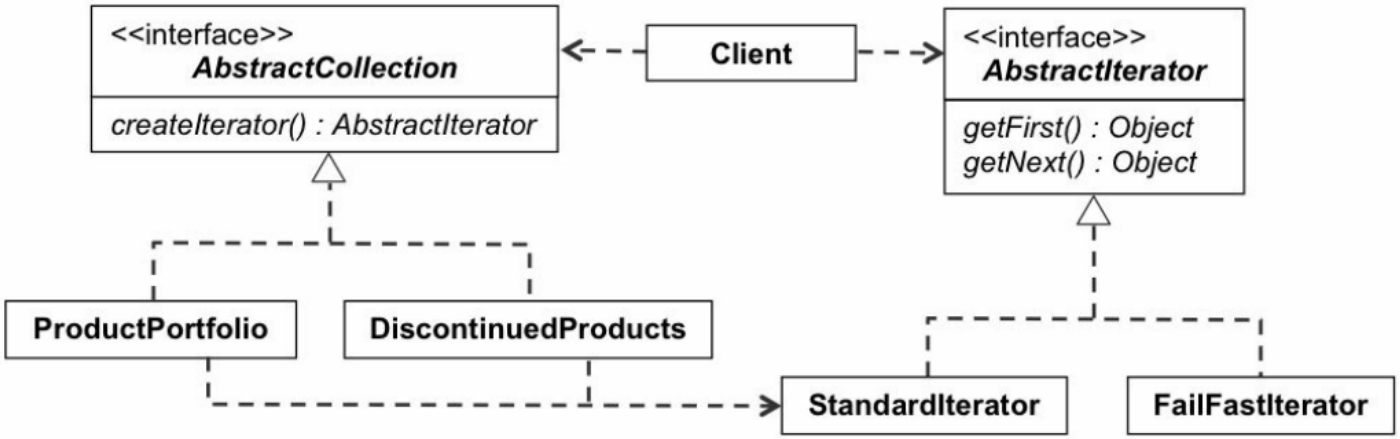
- Vorteile des neuen Designs:
  - Kohäsive Containerklasse
  - Möglichkeit, gleichzeitig mehrere Iteratoren auf demselben Container-Objekt zu haben
  - Möglichkeit, den Iterator für ähnliche Containerklassen wiederzuverwenden

- Aber: Unflexibles Design, denn:
  - Client-Code ist momentan eng an einen spezifischen Container gekoppelt (im Beispiel: `ProductPortfolio`)
  - Client-Code ist momentan eng an einen spezifischen Iterator gekoppelt (im Beispiel: `Iterator`)

Iterator: 3. Ansatz

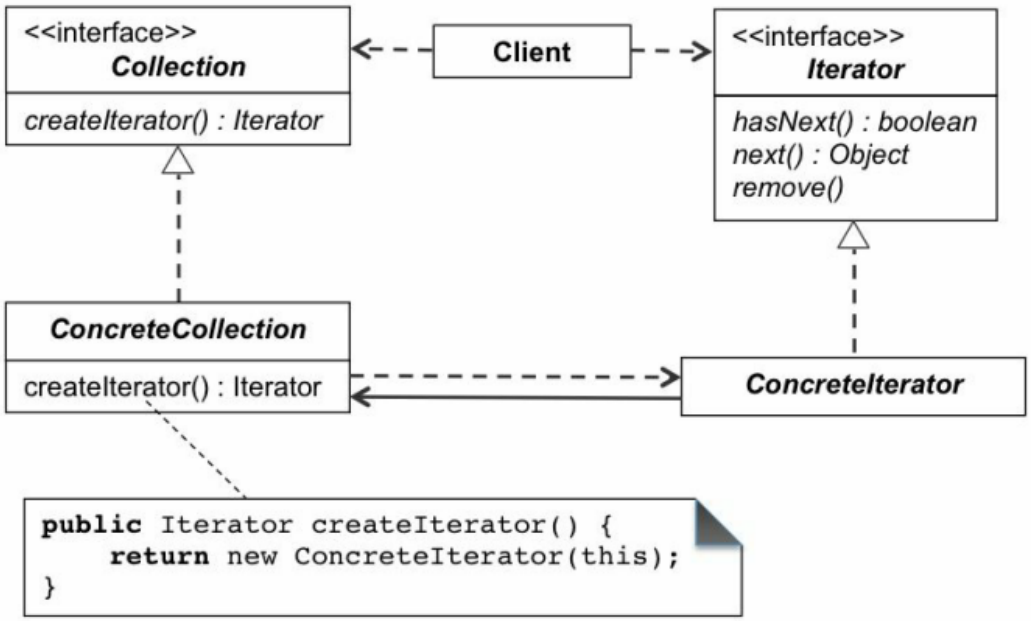
- **Lösung des obigen Problems:**
  - Verwendung von Interfaces für Container-Typen und Iterator-Klassen
  - Client-Code verwendet nur die abstrakten Interfaces

- **Konkretes Beispiel:**



Iterator: Lösung

- Abstrahiert man das obige Beispiel, so erhält man als Lösung folgendes Klassendiagramm:



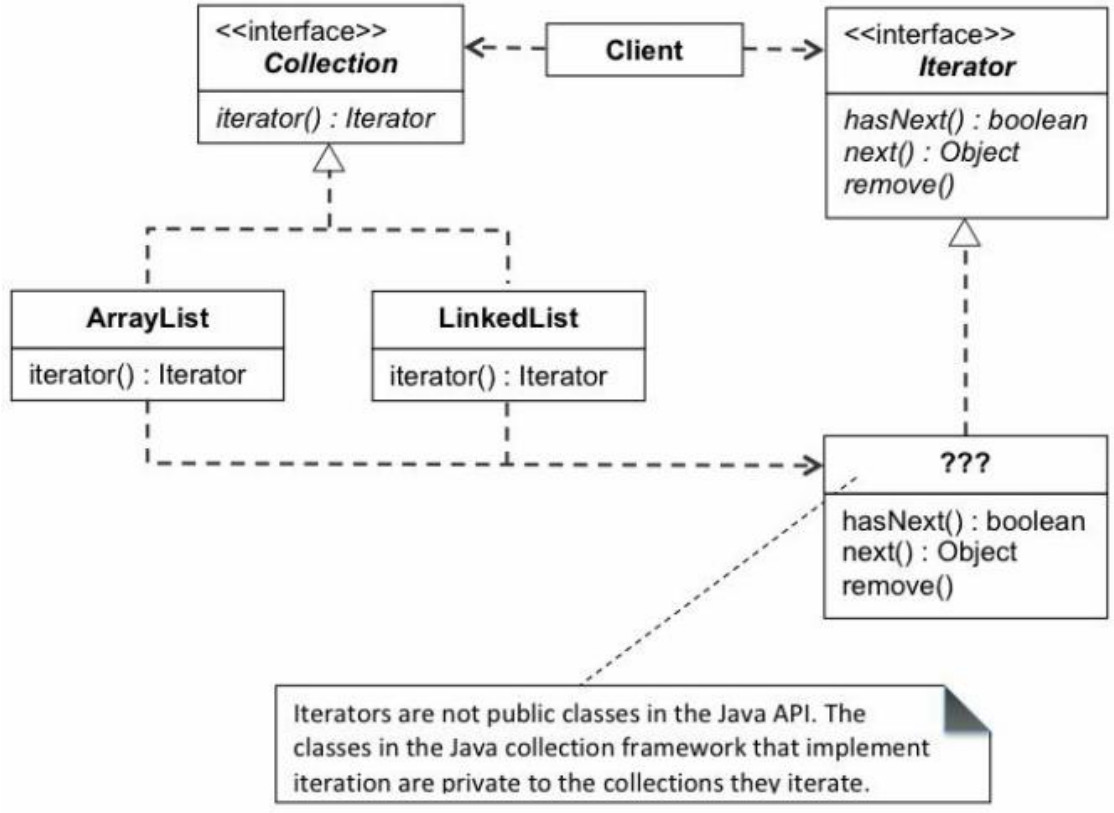
Iterator: Client-Code für die neue Lösung

Methode `clientCode(…)` ist unabhängig von konkreten Implementierungen

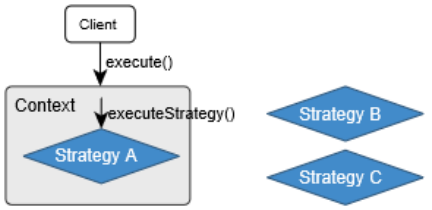
```
public static void main (String[] args) {

    DiscontinuedProducts dp = new DiscontinuedProducts();
    ProductPortfolio pp = new ProductPortfolio();

    clientCode(dp);
    clientCode(pp);
}
```

```
public static void clientCode(AbstractCollection c) {

    AbstractIterator iterator = c.createIterator();

    Product product = iterator.getFirst();

    while (product != null) {
        process(product);
        product = iterator.getNext();
    }
}
```

Iterator: Beispiel Java-Collection-Framework

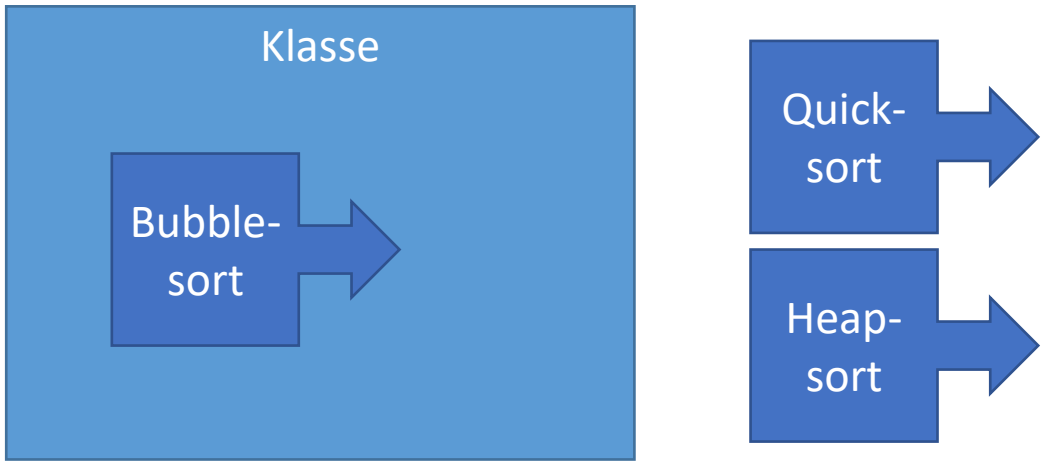


Strategy: Ziel und Idee (Einordnung: Verhaltensmuster)



[Bild: www.philipphauer.de]

- **Ziel: das Strategy-Pattern…**
  - Erlaubt es, verschiedene Verhalten/Algorithmen statt während des Designs erst zur Laufzeit zu nutzen oder zu ändern
- Beispiel:



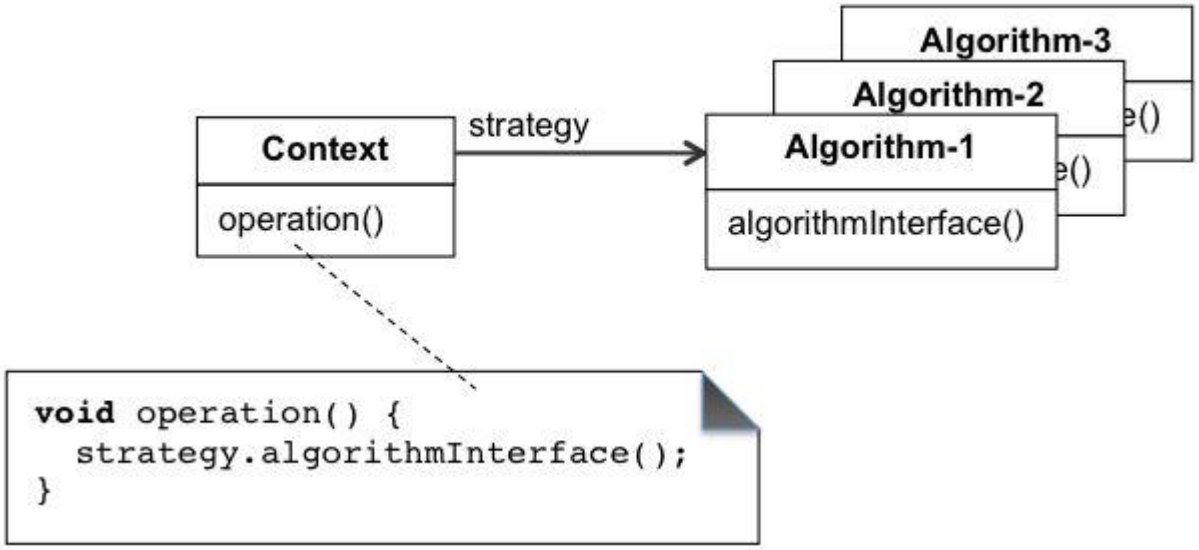
- Weitere Beispiele?
  -
  -

Strategy: Problemstellung und erste Lösungsansätze (1)

Wie kann ich Verhalten zur Laufzeit dynamisch austauschbar gestalten?

1. Ansatz:
   - Verschiedene Methoden pro Klasse anbieten

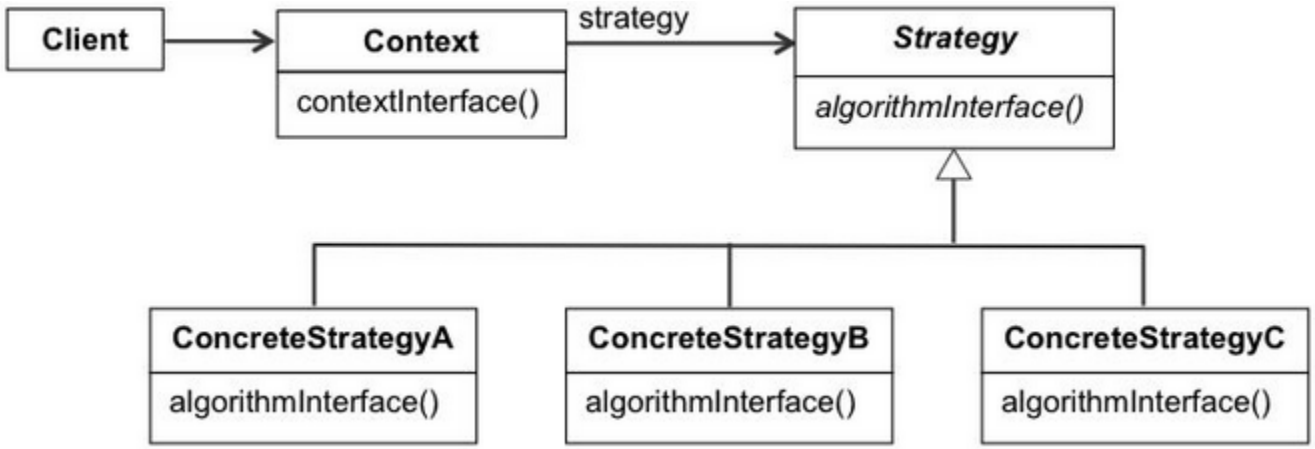
```
public class Context {
    void operation() {
        if (…) {
            // Algorithmus 1
        } else if (…) {
            // Algorithmus 2
        } else {
            // Algorithmus 3
        }
    }
    …
}
```



Strategy: Problemstellung und erste Lösungsansätze (2)

Wie kann ich Verhalten zur Laufzeit dynamisch austauschbar gestalten?

2. Ansatz:
   - Es gibt eine konkrete Klasse für jedes austauschbare Verhalten/Strategie
   - Abstrakte Strategie-Klasse mit abstrakter Methode, die die Schnittstelle für das gewünschte Verhalten definiert
   - Context ist eine Klasse mit konfigurierbarem Verhalten, die Referenz auf ein Objekt vom abstrakten Typ `Strategy` hält
   - Client kann üblicherweise das Verhalten über Context-Klasse wählen



Strategy: Allgemeines Code-Beispiel

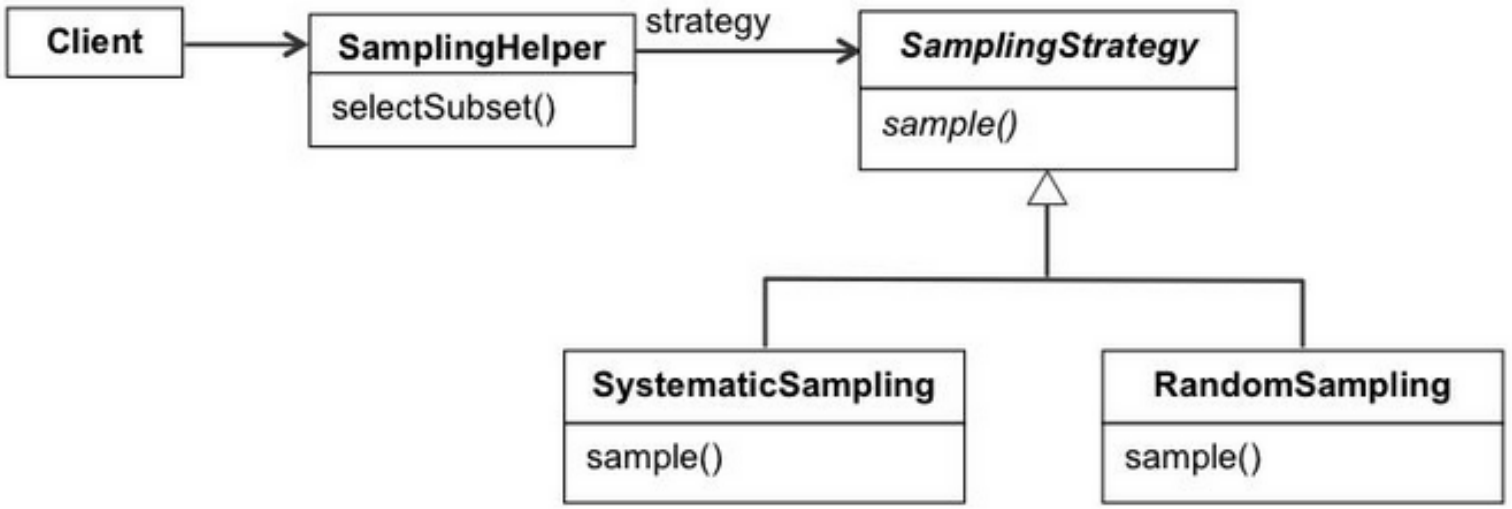
2. Ansatz als Codeauszug:

```java
public class Context {
    private Strategy strategy;

    public Context(Strategy s) {
        strategy = s;
    }

    public void contextInterface() {
        strategy.algorithm();
    }
}
```

```java
public class ConcreteStrategyB implements Strategy {

    @Override
    public void algorithm() {
        // implement concrete strategy B
        System.out.println("Executing algorithm of strategy B");
    }
}
```

```java
public class Client {

    void clientCode() {
        Strategy s = new ConcreteStrategyB();
        Context c = new Context(s);
        c.contextInterface();
    }

    public static void main (String[] args) {
        Client c = new Client();
        c.clientCode();
    }
}
```

Strategy: Konkretes Beispiel

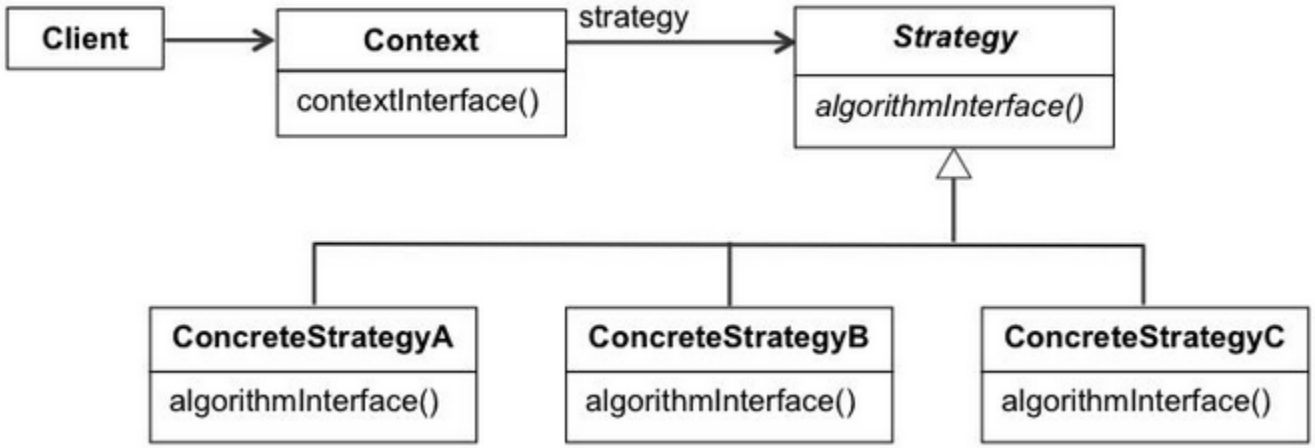


Strategy: Diskussion

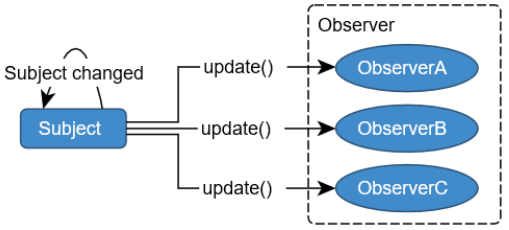
- Ein Problem, das oft bei der Umsetzung des Strategy-Patterns aufkommt:
  - Wie soll Datentransfer zwischen dem Kontext und dem Strategy-Objekt stattfinden?
- Lösungsmöglichkeiten:
  1. Der Kontext gibt Daten über Parameter an das Strategy-Objekt weiter
     - `strategy.sample(population, sampleSize)`
  2. Eine Referenz auf den Kontext wird weitergegeben und das Strategy-Objekt nutzt Callback-Methoden, um sich benötigte Daten zu beschaffen
     - `strategy.sample(this)`
- Bewertung der Lösungsmöglichkeiten:
  1. Lösungsmöglichkeit:
     - Bevorzugt, sofern alle Strategy-Objekte die gleiche Schnittstelle und nur wenige Parameter besitzen
  2. Lösungsmöglichkeit:
     - Bevorzugt, wenn Strategy-Objekte unterschiedliche/erhebliche Datenbedürfnisse haben
     - Nachteil: stärkere Kopplung (Abhängigkeit in zwei Richtungen); Kontext muss Datenbeschaffungsmethoden besitzen (Widerspruch zum Prinzip des Information-Hidings)

Strategy: Bewertung

- Bewertung des Musters:
  - Nachteil: Client muss Kenntnis der verschiedenen Strategies/Algorithmen haben
  - Dies erhöht die Kopplung zwischen Client und den Strategie-Implementierungen
  - Eine Abmilderungsmöglichkeit ist die Implementierung einer Default-Strategie

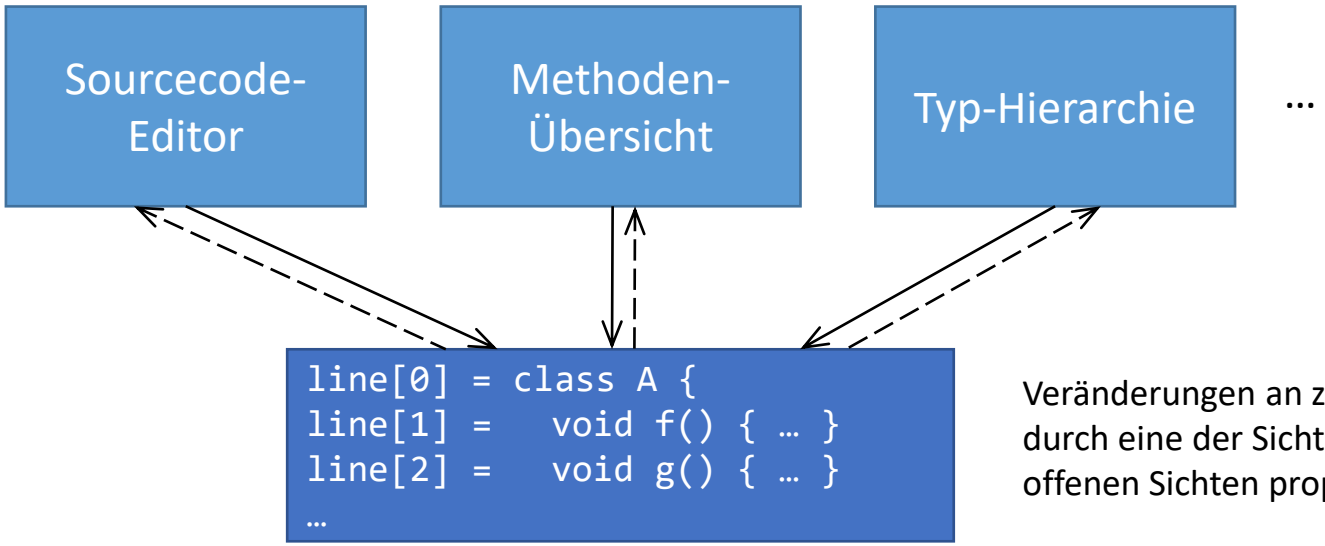


Observer: Ziel und Idee (Einordnung: Verhaltensmuster)



[Bild: www.philipphauer.de]

- **Ziel: das Observer-Pattern…**
  - Ist für Situationen geeignet, bei denen Objekte benachrichtigt werden müssen, sobald sich der Zustand eines zu beobachtenden Objektes ändert
- Beispiel: eine IDE (z.B. IntelliJ, Visual Studio etc.)



Veränderungen an zugrundeliegenden Daten durch eine der Sichten wird in die anderen offenen Sichten propagiert

- Weitere Beispiele?
  -
  -

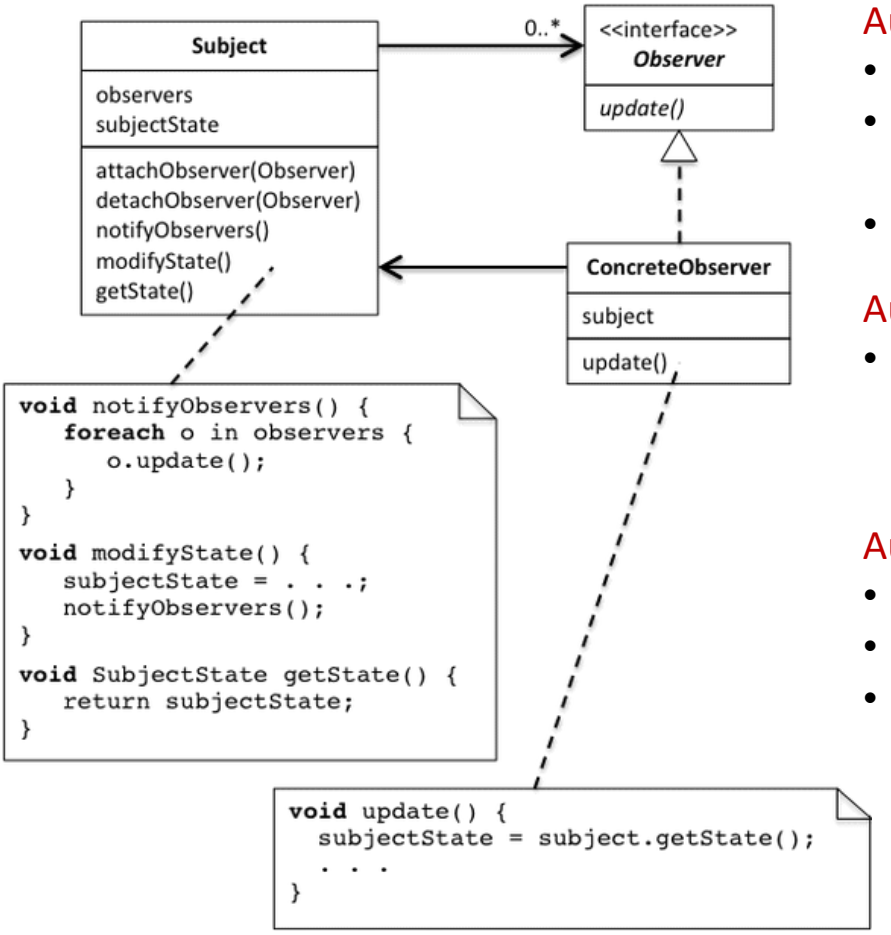
Observer: Problemstellung und erste Lösungsansätze

Wie kann ich interessierte Objekte über Veränderungen informieren?

1. Alle Informationen in einer großen, eng-gekoppelten Klasse vorhalten
   - Macht Wiederverwendung der Sichten oder Datenstruktur unmöglich
   - Erschwert Wartung (Hinzufügen neuer Sicht macht Modifikation existierender Klasse notwendig)

2. Sichten und darunterliegende Datenstruktur werden in getrennten, lose-gekoppelten, kooperierenden Objekten organisiert
   - Erleichtert Wiederverwendung und Erweiterbarkeit
   - Problem: wie hält man einzelne Sichten mit darunterliegenden Daten synchron?
     - Sichten-Objekte stellen wiederholt Anfragen (polling) an die Datenstruktur → Verschwendung von Rechenressourcen
     - Subjekt-Objekt verwaltet eine Liste aller zu informierenden Objekte und informiert diese, sobald neue Informationen vorliegen

Observer: Lösung



Aufgaben des Subjekts:

- Hinzufügen und Entfernen von Observern
- Zugeordnete Observer über Zustandsänderungen des Subjekts informieren
- Umsetzen applikationsspezifischer Logik

Aufgaben des Observers:

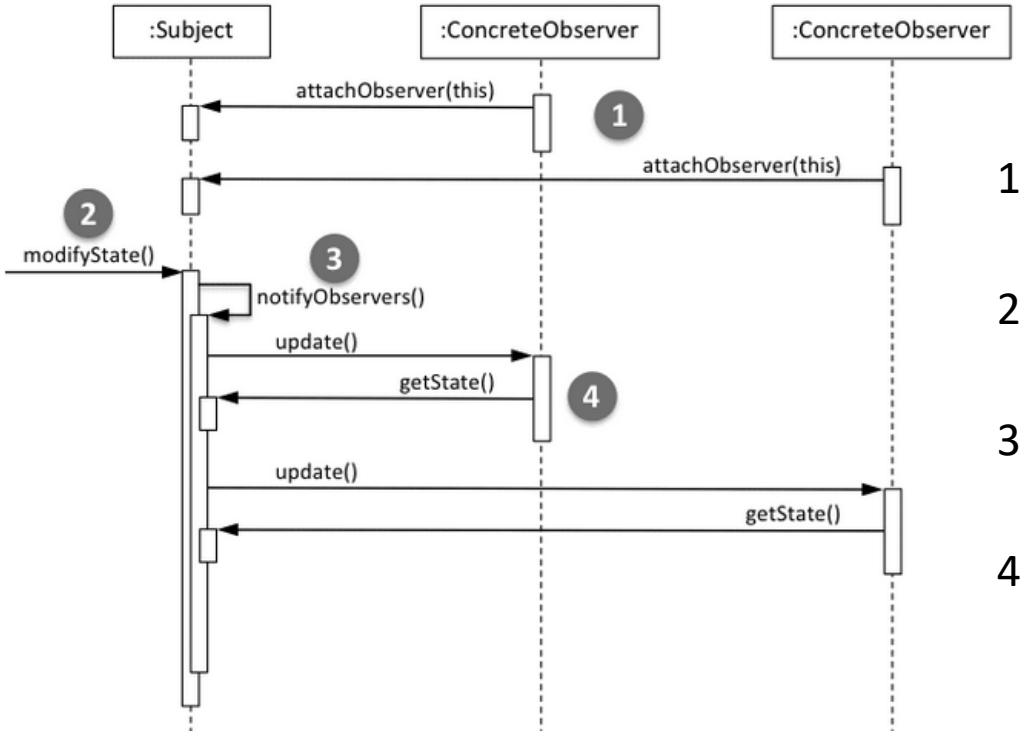
- Definiert abstrakte Schnittstelle mit einer Callback-Methode (hier: `update()`) zur Benachrichtigung von Observer-Objekten

Aufgaben des konkreten Observers:

- Implementiert die abstrakte Schnittstelle Observer
- Kann eine Referenz auf das Subjekt haben
- Referenz wird benötigt, um weitere Informationen bei Benachrichtigung über Zustandswechsel vom Subjekt abzurufen

Vorteile dieses Designs?

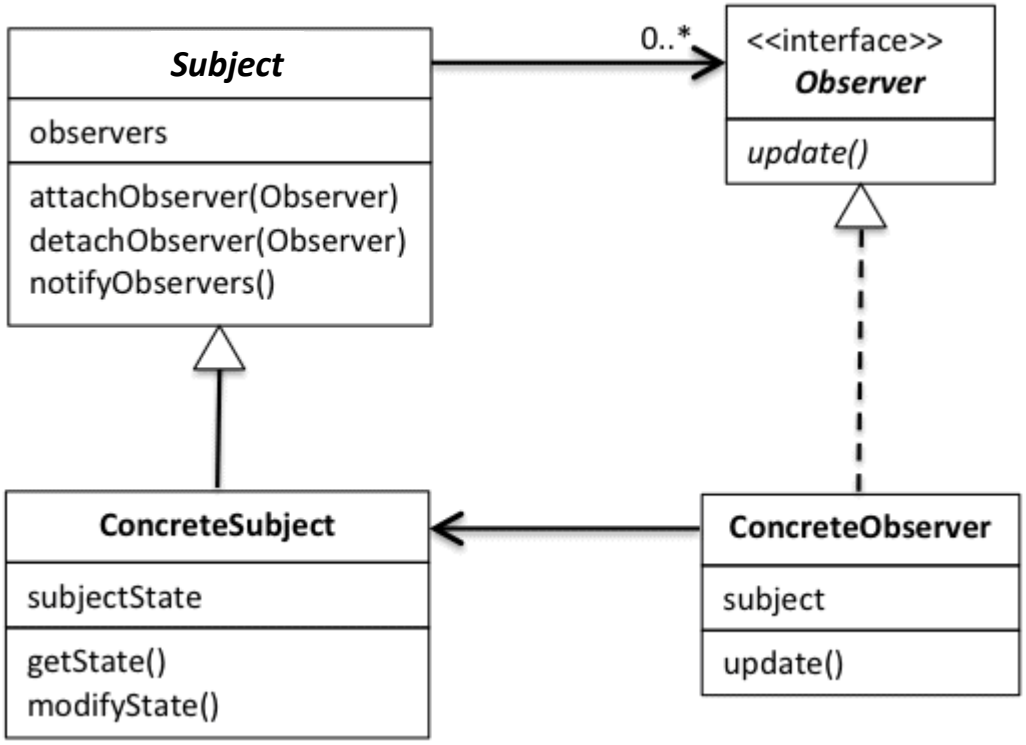
Observer: Lösung
Typische Interaktion



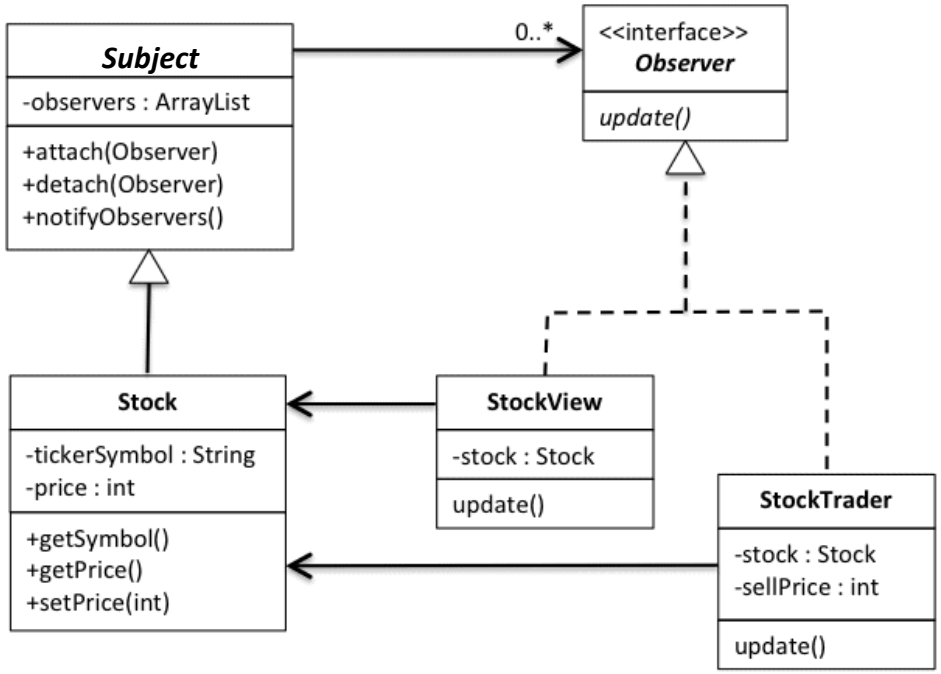
1. Observer werden (aktiv oder passiv) beim Subjekt registriert
2. Eine weitere Komponente bewirkt eine Zustandsänderung beim Subjekt
3. Nach Zustandsänderung werden alle registrierten Observer informiert
4. Observer können beim Subjekt zusätzliche Informationen über Änderung erfragen

Welche Modellierungs-konventionen sind verletzt?

Observer: Finale Lösung



Observer: Konkretes Beispiel



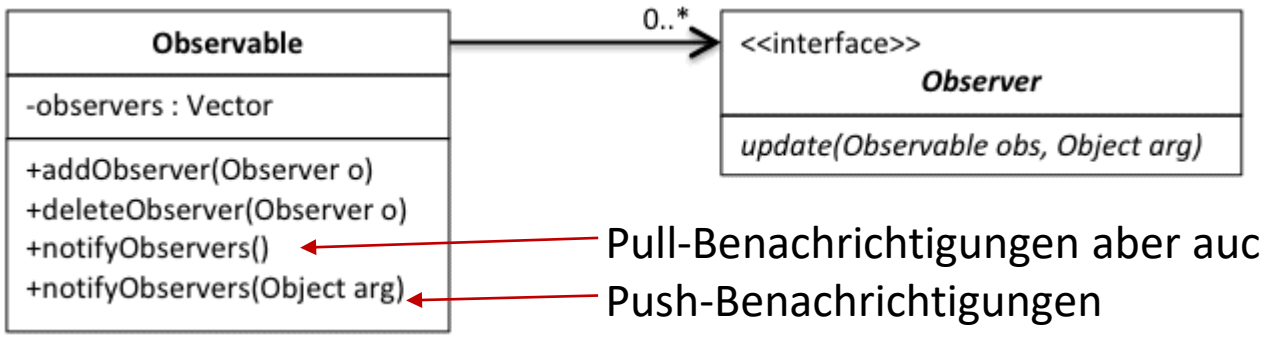

```java
public class OberverRunner {

    public static void main (String[] args) {
        Stock stock = new Stock("Alphabet Inc", 1198);
        StockView view = new StockView(stock);
        StockTrader trader = new StockTrader(stock, 1200);

        stock.setPrice(1199);
        stock.setPrice(1200);
    }
}
```

```
Console  Tasks  Search  JUnit  Coverage
<terminated> OberverRunner [Java Application] C:\Program Files\ojdkbuild\java-11-openjdk-11.0.3-1\bin\javaw.exe
UPDATE: Alphabet Inc is now selling for 1199
UPDATE: Alphabet Inc is now selling for 1200
INTERESTING UPDATE: Sell Alphabet Inc!
```

Observer: Diskussion

In vielen Programmiersprachen wird das Observer-Muster schon bereitgestellt

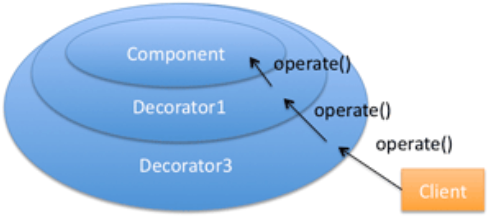
- Beispiel aus dem Java JDK (bis Java 8):



Bewertung des Musters:

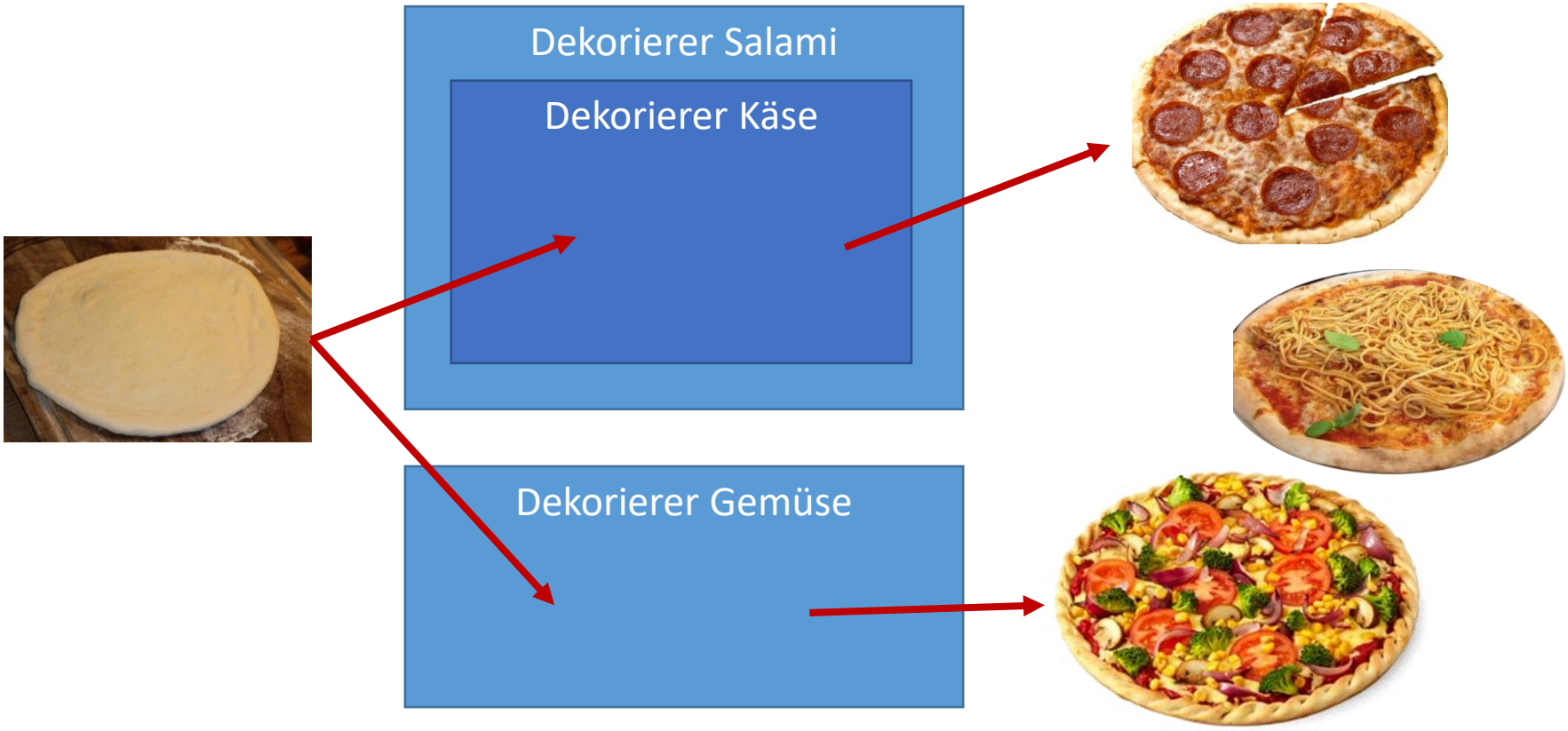
- Lose Kopplung zwischen Subjekt und seinen Observern (durch abstrakte Schnittstelle)
- Kein Leistungsverlust durch wiederholtes Anfragen der Observer nach Zustandsänderungen
- Dynamisches Informieren von Interessenten bei Zustandsänderungen von Objekten möglich

Decorator: Ziel und Idee (Einordnung: Strukturmuster)



[Bild: www.philipphauer.de]

- **Ziel: das Decorator Pattern…**
  - erlaubt es, flexibel das Verhalten bestehender Klasse zu erweitern



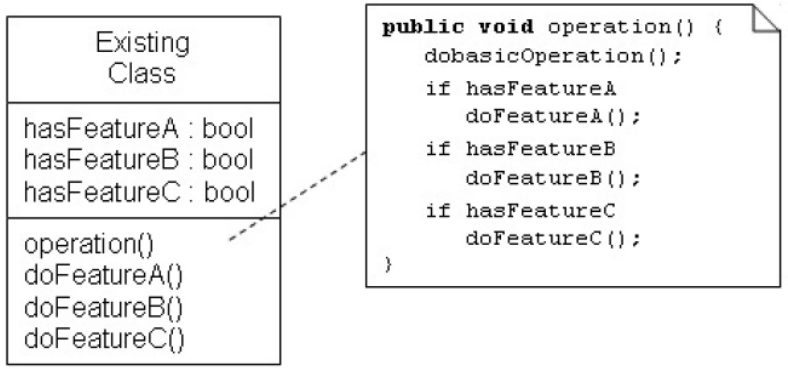
Decorator: Problemstellung und erste Lösungsansätze

Wie kann ich das Verhalten einer Klasse erweitern?

1. Code der Klasse ändern

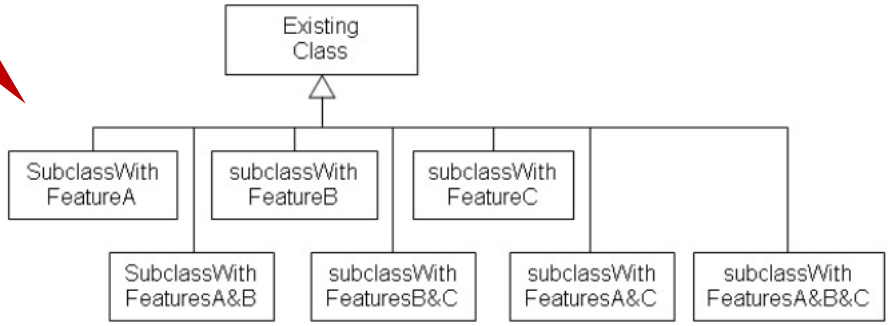
   → Open-Close-Prinzip verletzt

   (Klassen sollen für Erweiterungen offen aber Modifikationen geschlossen sein)



2. Erweiterung durch Vererbung

   → Potentielle Explosion der Anzahl an Klassen bei Kombination von Features



Decorator: Beispiel

Ein Online-Store möchte Ebook-Reader in drei Versionen anbieten:

- Standard
- Groß
- Tablet

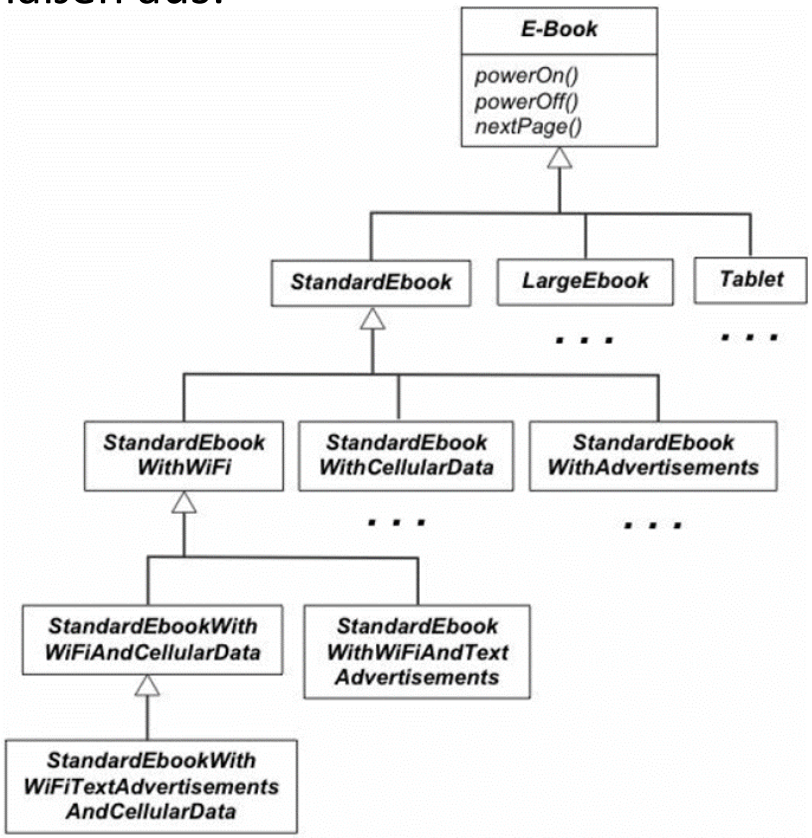
Außerdem soll es zu jedem Modell Kombinationen der folgenden Features geben:

- mit WIFI
- mit 5G-Wireless
- mit Werbeeinblendung (30 Euro Subventionierung)

Wie viele Varianten macht das?

Decorator: Beispiel

Für unser Beispiel sähe die Realisierung über Vererbung der Features etwa folgendermaßen aus:

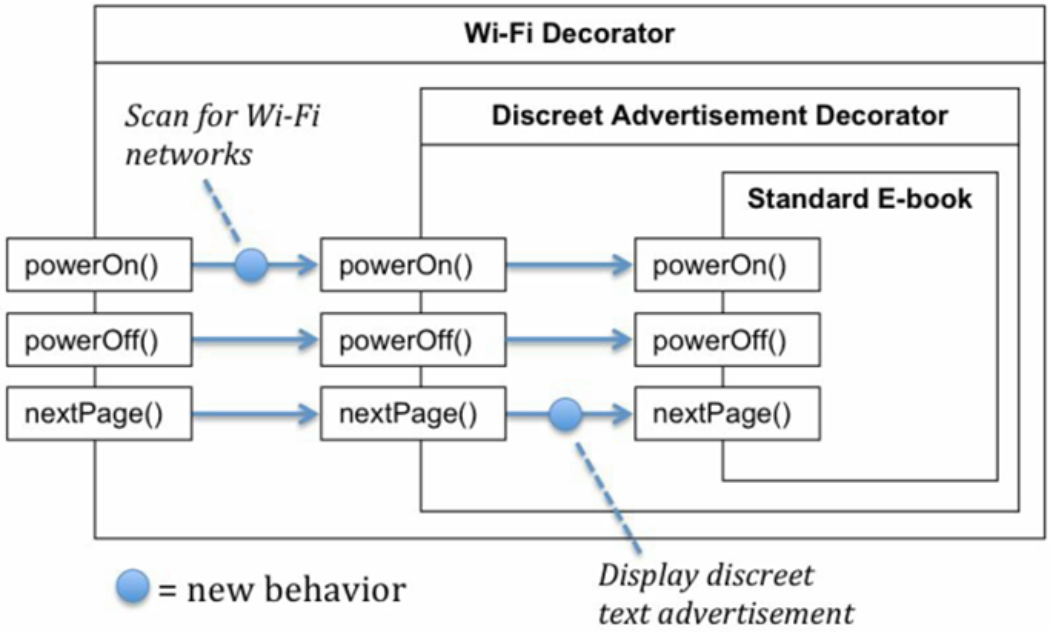


eine solche Lösung ist ein Wartungsalbtraum!

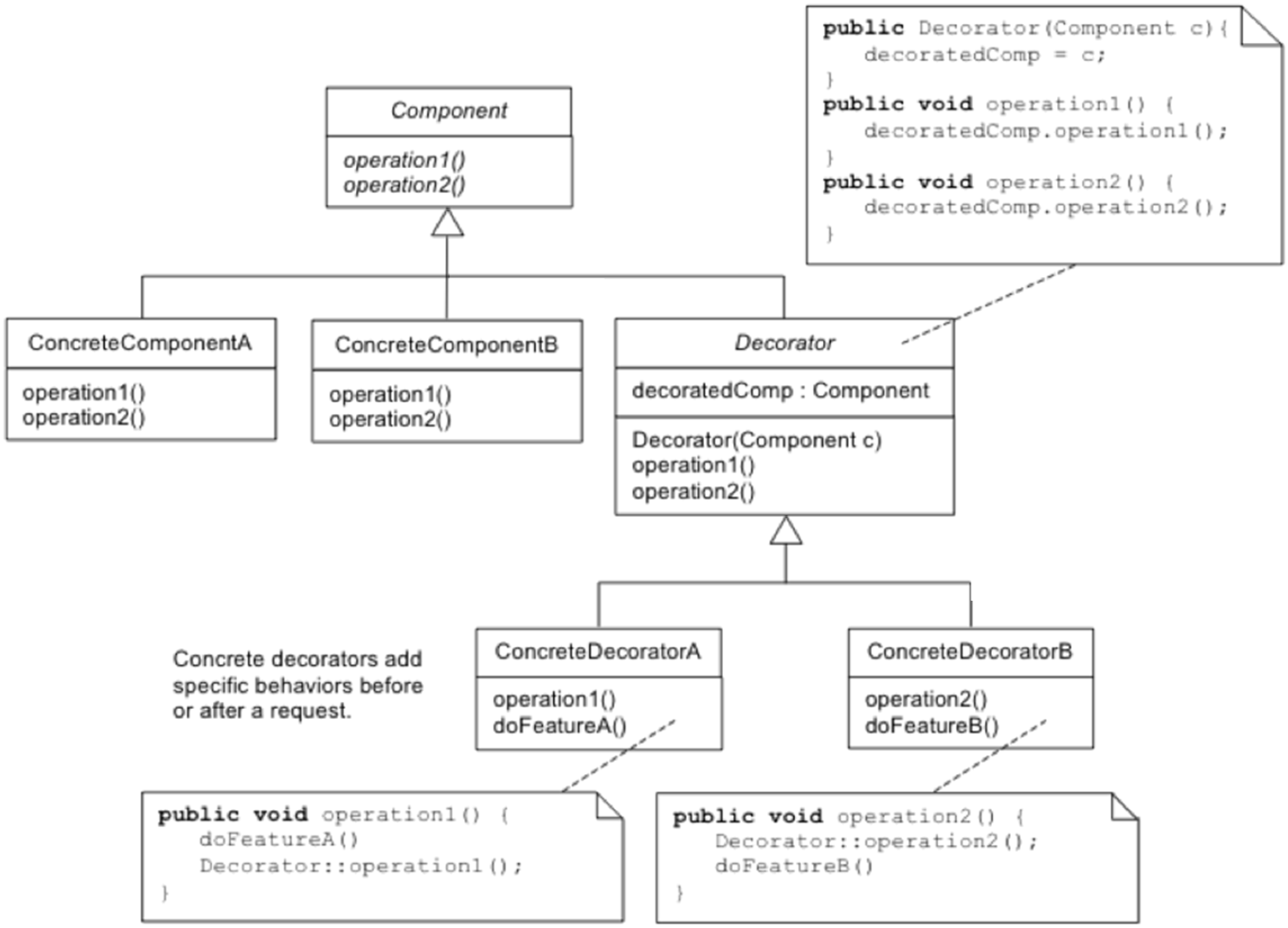
Decorator: Lösungsidee

- **Idee:**
  - Es gibt 3 Basismodelle des Ebook-Readers (standard, groß, Tablet) und
  - Verschiedene “Folien” für die einzelnen Features

- **Das führt uns zum Decorator-Pattern:**



Decorator: Lösung



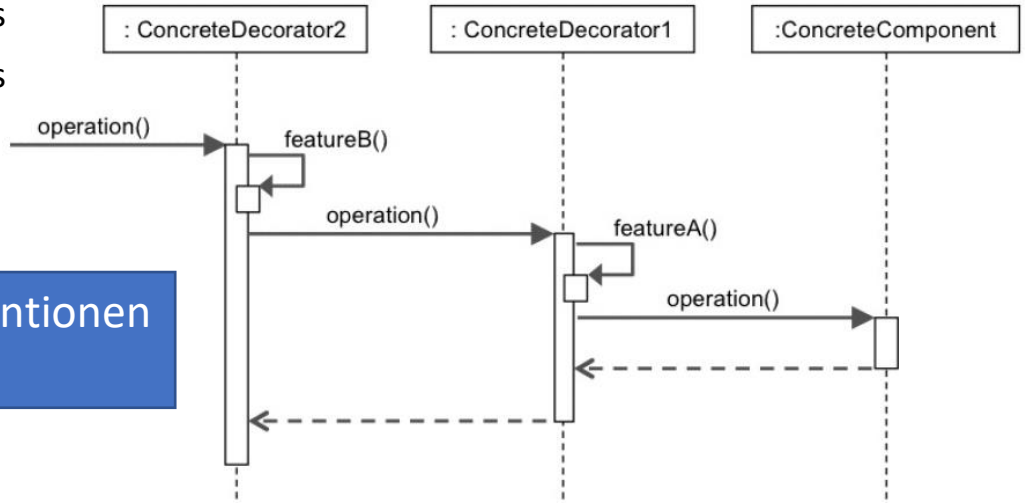
Decorator: Lösung

- **Um einer Komponente ein Feature hinzuzufügen:**
  - Erzeugt man Instanz der Komponenten und
  - Übergibt diese Instanz dem Konstruktor des konkreten Decorators

```
ConcreteComponent  cc  = new ConcreteComponent();
ConcreteDecorator1 cd1 = new ConcreteDecorator1(cc);
ConcreteDecorator2 cd2 = new ConcreteDecorator2(cd1);
cd2.operation();
```

`ConcreteDecorator2` führt `featureB()` aus

`ConcreteDecorator1` führt `featureA()` aus



Welche Konventionen sind verletzt?

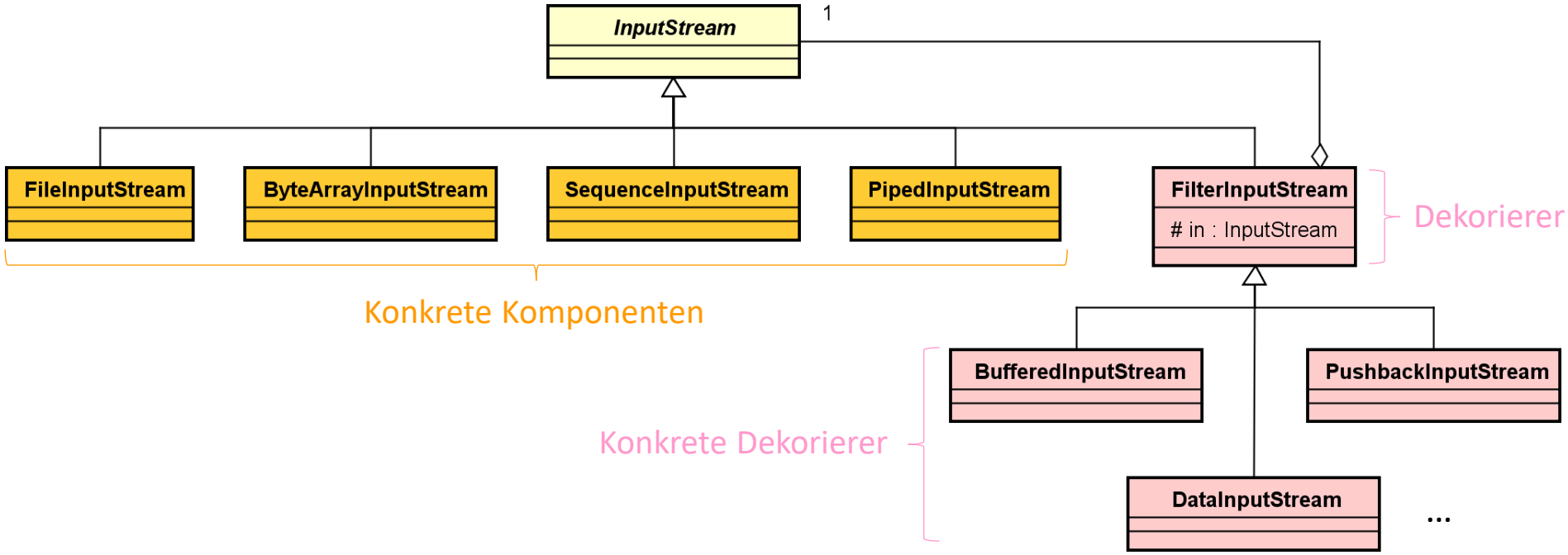
Decorator: Beispiel

Wie sieht das Klassendiagramm für unser Beispiel mit den Ebook-Readern aus?

Decorator: Beispiel

Wie sieht der Code für das Erstellen, Dekorieren und die Nutzung aller Funktionalitäten eines StandardEbooks mit Wifi und Werbung aus?

Decorator: weiteres Beispiel (Java IO)



- `FileInputStream`: *"obtains input bytes from a file in a file system …"*
- `BufferedInputStream`: *"adds functionality to another input stream-namely, the ability to buffer the input …"*
- `GZIPInputStream`: *"implements a stream filter for reading compressed data in the GZIP file format …"*

- Über das Muster Dekorierer kann man in Java so also sehr einfach z.B. den Inhalt einer gepackten Datei mithilfe der obigen drei Stream-Klassen ausgeben. Wie sähe eine Implementierung aus?

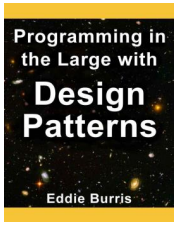
Zusammenfassung: Design Patterns

Wichtige Prinzipien von Design Patterns:

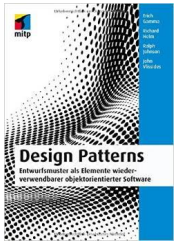
- Programmiere gegen ein Interface, nicht gegen die Implementierung
- Ziehe Objekt-Komposition der Vererbung vor
- Koppele Objekte lose
- Kapsele variable Konzepte

Literatur

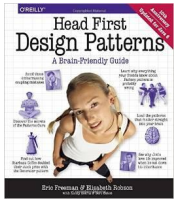
- Programming in the Large with Design Patterns, E. Burries, Pretty Print Press 2012



- Entwurfsmuster – Elemente wiederverwendbarer objektorientierter Software, E. Gamma et al., Addison-Wesley, 2004



- Head First Design Patterns, E. Freeman et al., O'Reilly, 2004



- http://www.philipphauer.de/study/se/design-pattern.php