

2013

Name: Osman Özsoy
Klasse: 4AHIT
Datum: 13.12.2013



[DESIGN-PRINZIPIEN]

Inhaltsverzeichnis

Inhaltsverzeichnis	1
Aufgabenstellung	2
Quellen	2
Design Patterns	3
Strategy Pattern.....	3
UML-Klassendiagramm.....	3
Design Prinzipien	4
Observer Pattern	6
UML-Klassendiagramm.....	6
Design Prinzipien	7
Decorator Pattern	8
UML-Klassendiagramm.....	9
Design Prinzipien	9
Factory Pattern	10
UML-Klassendiagramm.....	11
Design Prinzipien	12
Adapter Pattern	13
UML-Klassendiagramm.....	13
Design Prinzipien	14
Iterator und Composite Pattern	16
UML-Klassendiagramm.....	17
Design Prinzipien	18
Quellenangaben	19

Aufgabenstellung

Schreibe eine Zusammenfassung der im Buch "Head First - Design Patterns" vorgestellten Design Prinzipien [1]. Gebe dabei immer die verwendeten Patterns an (Name, UML-Diagramm, Kurzbeschreibung) und zeige die wichtigsten Vor- als auch Nachteile auf. Es ist auch von Vorteil auf mögliche Codebeispiele zu verweisen [2].

Gebe diese Aufgabenstellung als PDF-Dokument ab. Bedenke, dass dieses Protokoll zu allererst dir von großem Nutzen sein soll! Spare nicht bei der Referenzierung deiner Quellen!

Quellen

[1] Herauskopierte Design Prinzipien aus "Head First Design Patterns; Eric Freeman, Elisabeth Freeman, Kathy Sierra, Bert Bates; First Edition; 2004; O'Reilly Media"; Online: <http://elearning.tgm.ac.at/mod/resource/view.php?id=20001>; abgerufen 12.12.2013

[2] Codebeispiel zur Abstract Factory in C++; Vince Huston; 2007; Online: <http://www.vincehuston.org/dp/FactoryDemosCpp>; abgerufen 12.12.2013

Design Patterns

Gründe für die Verwendung von Design Patterns

- **Sourcecode**
 - leichter zu implementieren
 - einfacher zu erweitern
 - Wartbarkeit
- **Programmierung**
 - Steigerung der Effizienz
 - Software-Design
 - Qualität der Projekte

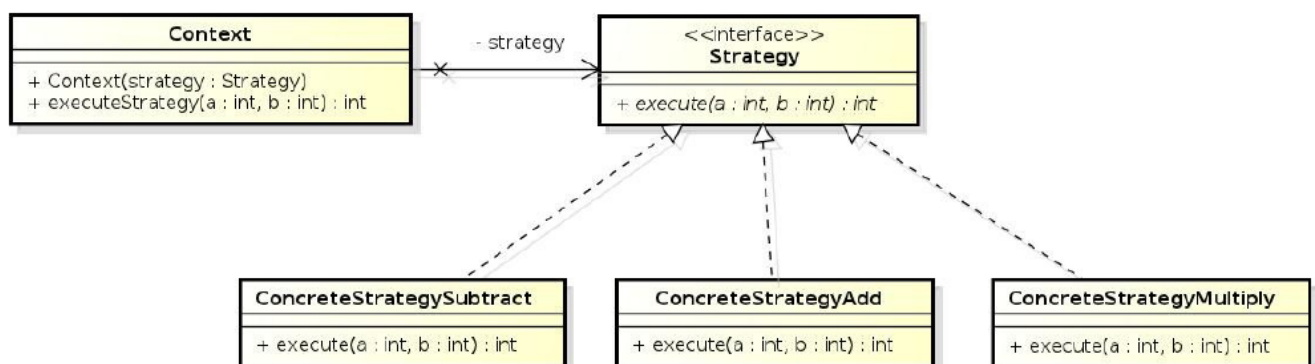
Basis von Design Patterns

- Objekt-orientiertes Paradigma
- Lösungen zu Programmierproblemen
- Gute Design-Techniken
- Erste Veröffentlichung von der **Gang of Four**

Strategy Pattern

- Das Strategy Pattern definiert eine Familie von Algorithmen, kapselt sie einzeln und macht sie austauschbar.
- Algorithmus unabhängig von Clients variierbar.
- **Link für ein Java-Codebeispiel:** <http://www.fluffycat.com/Java-Design-Patterns/Strategy/>

UML-Klassendiagramm

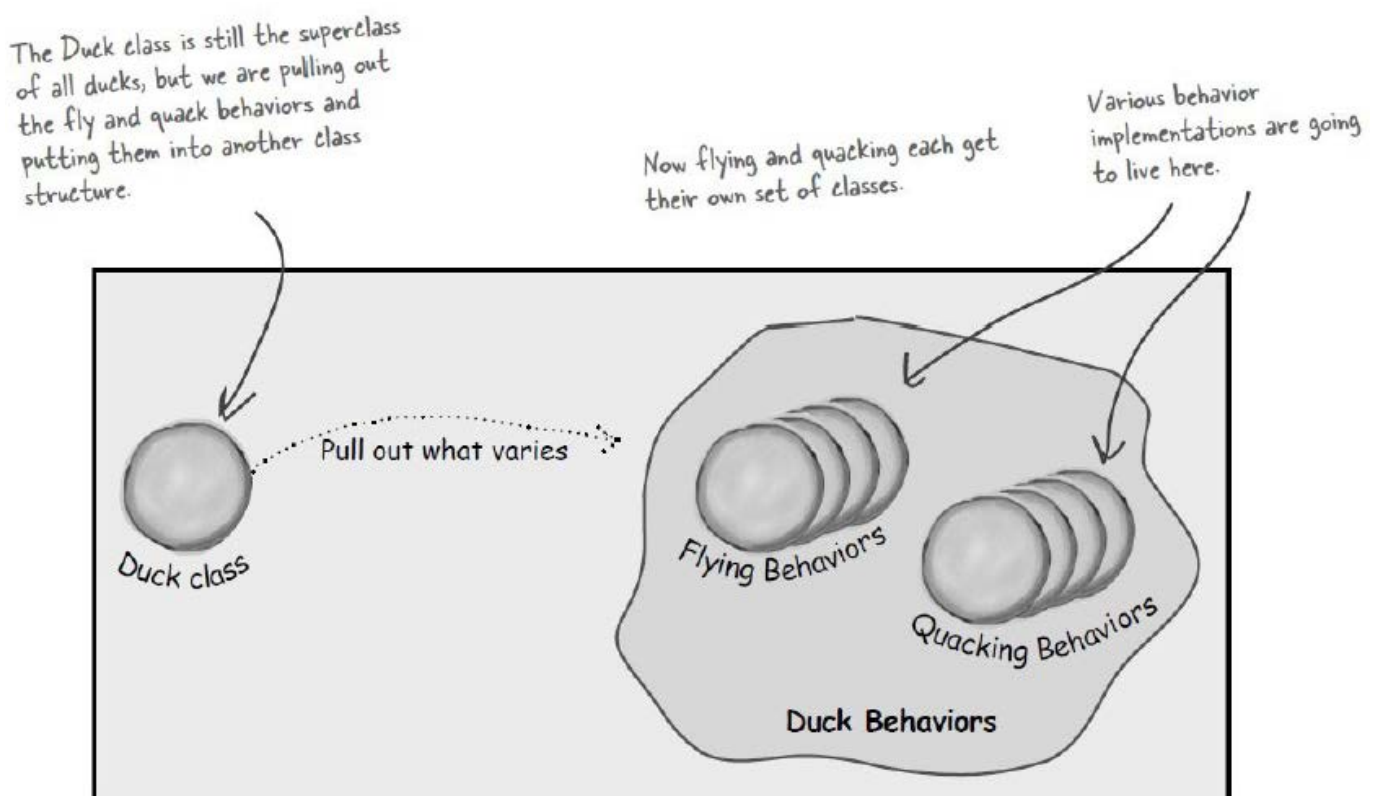


Design Prinzipien

Prinzip 1

Identifiziere die Aspekte deiner Anwendung, die variieren und trenne sie von dem, was gleich bleibt.

- **Problem:**
 - Codeverdupplung
 - Immer, wenn eine Verhaltensänderung gebraucht wird, muss in den verschiedenen Unterklassen, wo das Verhalten definiert ist, geändert werden. Hier werden neue Fehler eingeführt in den Code.
- Mit anderen Worten gesagt: Nimm das was variiert und kapsle sie so, dass es den Rest des Codes nicht beeinflusst.
- **Lösung: Extrahieren des Verhaltens:**



Prinzip 2

Programmiere auf Interfaces und nicht auf Implementierungen.

- **Schnittstelle:**
 - Interface
 - Abstrakte Klasse
- **Auf ein Interface implementieren = auf ein Supertyp implementieren**
- Auf eine Implementierung programmieren:

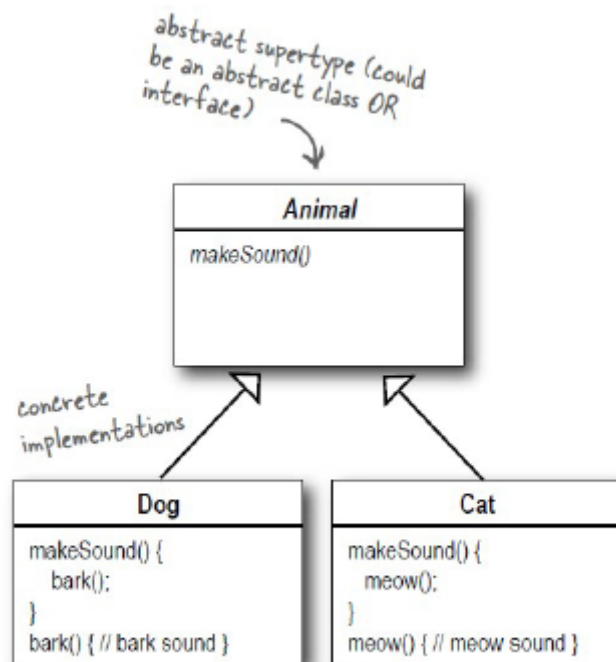
```
Dog d = new Dog();  
d.bark();
```

- Auf eine Schnittstelle/Supertyp programmieren:

```
Animal animal = new Dog();  
animal.makeSound();
```

- Implementierung zur Laufzeit zuweisen:

```
a = getAnimal();  
a.makeSound();
```



Prinzip 3

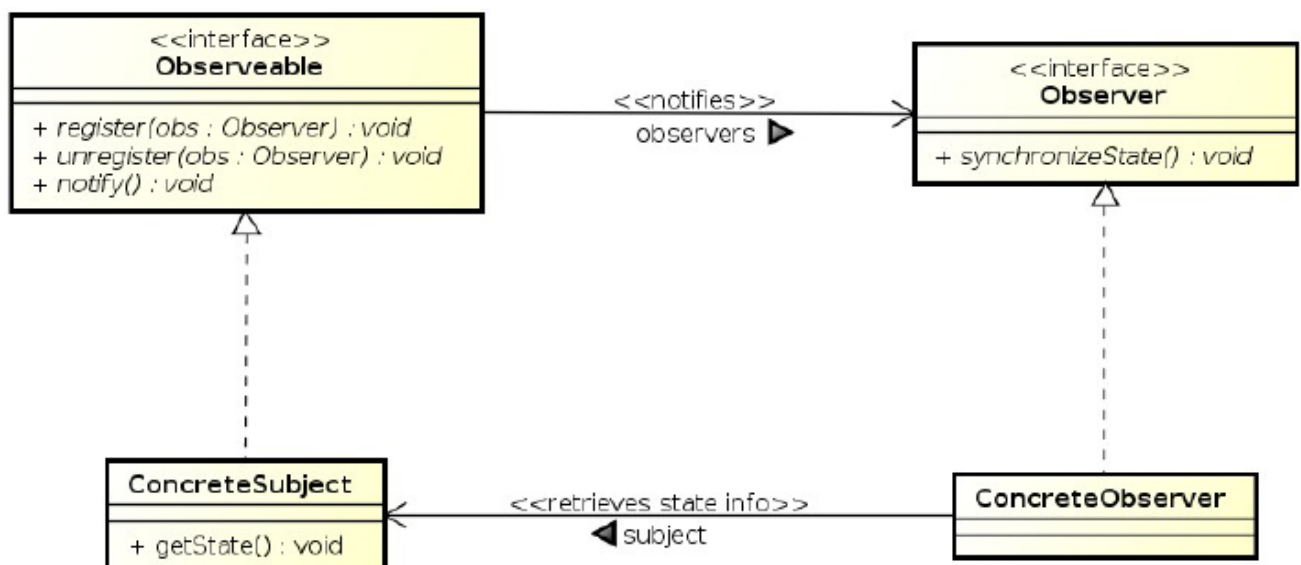
Begünstige Komposition der Vererbung.

- HAS-A kann besser sein als IS-A
- HAS-A ist eine interessante Beziehung: z.B.: Jede Ente hat ein (has a) FlyBehavior und ein QuackBehavior zu denen es fliegen und quaken delegiert.
- Wenn du zwei Klassen wie dies zusammensetzt, benutzt du eine Komposition.
- Anstatt ihr Verhalten zu erben, bekommen die Enten ihr Verhalten durch das Zusammensetzen mit dem richtigen Verhalten Objekt.
- Erzeugen von Systemen unter Benutzung der Komposition gibt viel mehr Flexibilität und lässt das Verhalten während der Laufzeit verändern.

Observer Pattern

- Das Observer Pattern definiert eine „Eins-zu-viele-Abhängigkeit“ zwischen Objekten in der Art, dass alle abhängigen Objekte benachrichtigt werden, wenn sich der Zustand des einen Objekts ändert.
- **Link für ein Java-Codebeispiel:** <http://www.fluffycat.com/Java-Design-Patterns/Observer/>

UML-Klassendiagramm

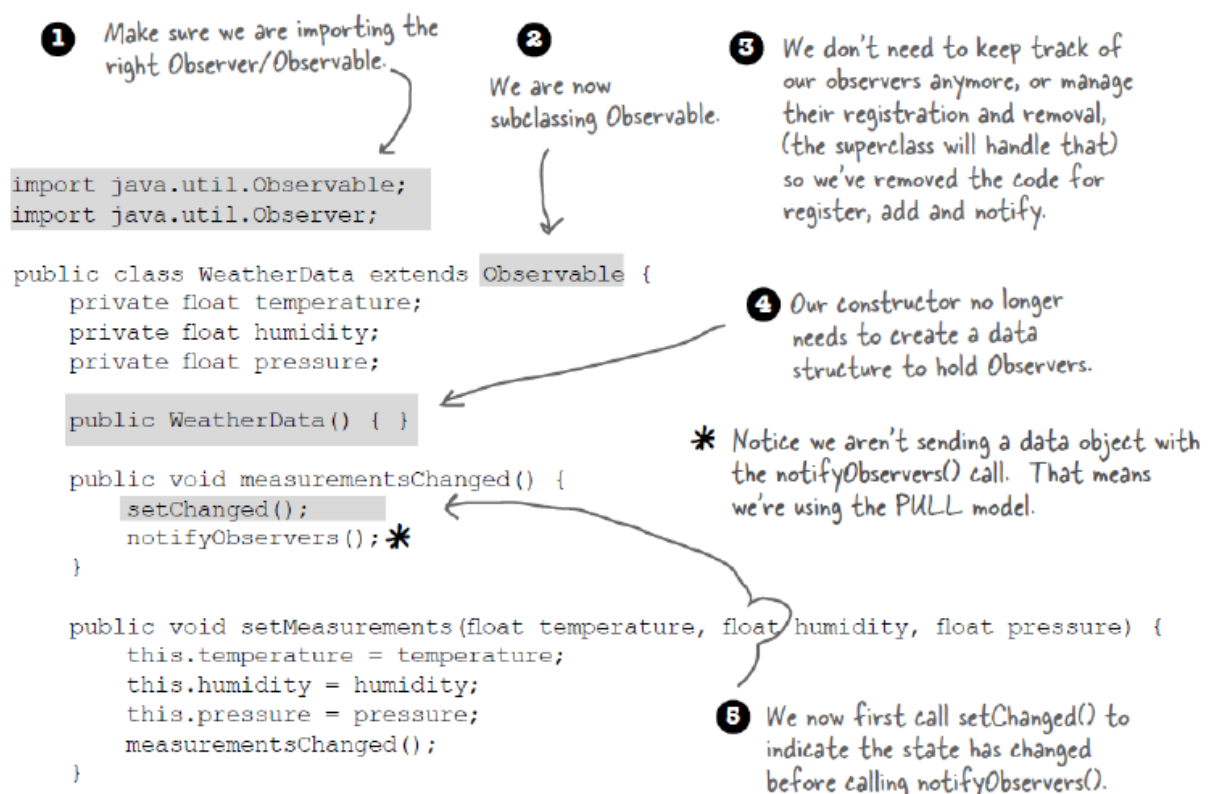


Design Prinzipien


Prinzip

Strebe für lose gekoppelte Designs zwischen Objekten, die interagieren.

- Locker gebunden = Interaktion mit wenig Detailwissen
- Lose gekoppelte Designs ermöglichen das Bauen von flexiblen OO Systemen, die die Änderung bewältigen können, weil sie die wechselseitige Abhängigkeit minimieren.
- **Observer Pattern: Lockere Kopplung zwischen Subjekt und Beobachter:**
 - Subjekt kennt von einem Beobachter nur die Beobachter-Schnittstelle
 - Subjekt muss für neue Beobachter nicht verändert werden
 - Subjekt und Beobachter sind unabhängig verwendbar
- **Funktionsweise:**
 - Beobachter-Klassen implementieren `java.util.Observer`
 - Die Subjekt-Klasse erweitert `java.util.Observable`
 - Nachrichten schicken:
 - ♦ `setChanged()` aufrufen
 - ♦ `notifyObservers()` oder `notifyObservers(Object arg)`
 - Benachrichtigung erhalten:
 - ♦ `Update(Observable o, Object arg)` implementieren




```
public float getTemperature() {  
    return temperature;  
}  
  
public float getHumidity() {  
    return humidity;  
}  
  
public float getPressure() {  
    return pressure;  
}  
}
```

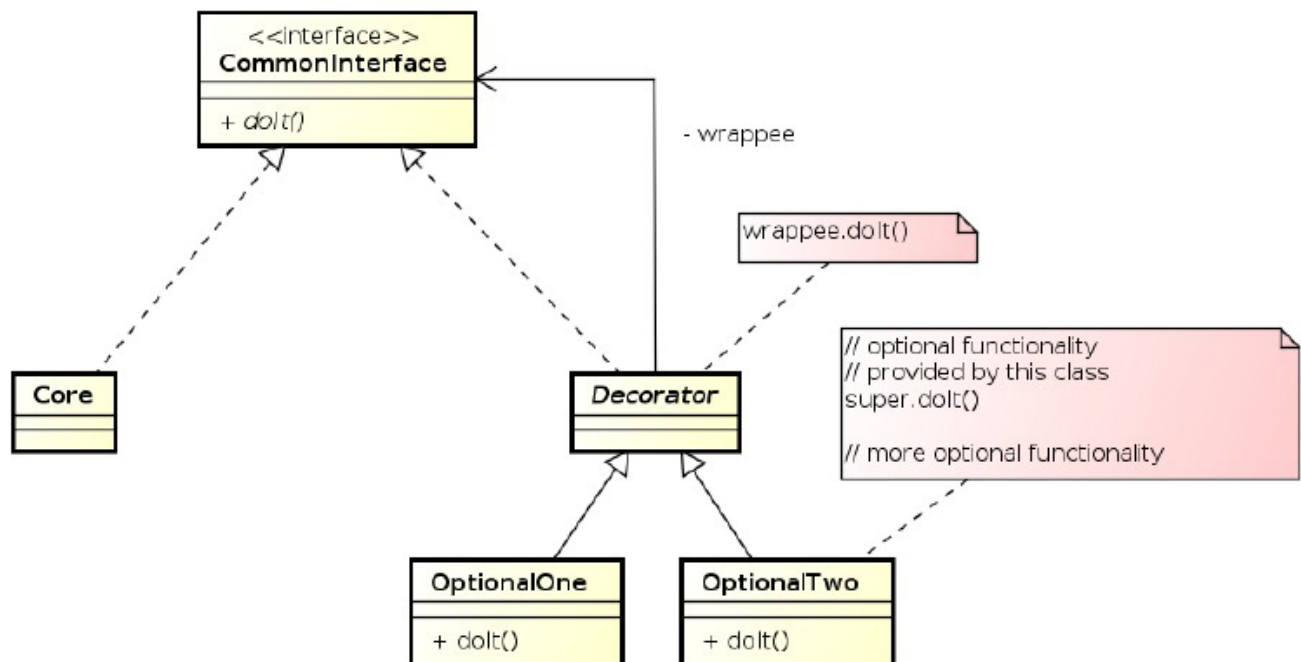


6 These methods aren't new, but because we are going to use "pull" we thought we'd remind you they are here. The Observers will use them to get at the WeatherData object's state.

Decorator Pattern

- Das Decorator Pattern definiert eine flexible Alternative zur Unterklassenbildung, um eine Klasse um zusätzliche Funktionalitäten zu erweitern.
- Dekorierer-Klassen werden verwendet, um konkrete Komponenten einzupacken
- Dekorierer-Klassen spiegeln den Typ der Komponente wider, die sie dekorieren
- Dekorierer ändern das Verhalten der Komponenten, indem sie vor und/oder nach Methodenaufrufen auf der Komponente neue Funktionalitäten hinzufügen
- Man kann eine Komponente mit einer beliebigen Anzahl von Dekorierern einpacken
- Dekorierer sind für die Clients der Komponente üblicherweise transparent
- Dekorierer können zu vielen kleinen Objekten führen -> übermäßige Verwendung -> unübersichtlicher Code.
- **Link für ein Java-Codebeispiel:** <http://www.fluffycat.com/Java-Design-Patterns/Decorator/>

UML-Klassendiagramm



Design Prinzipien

Prinzip

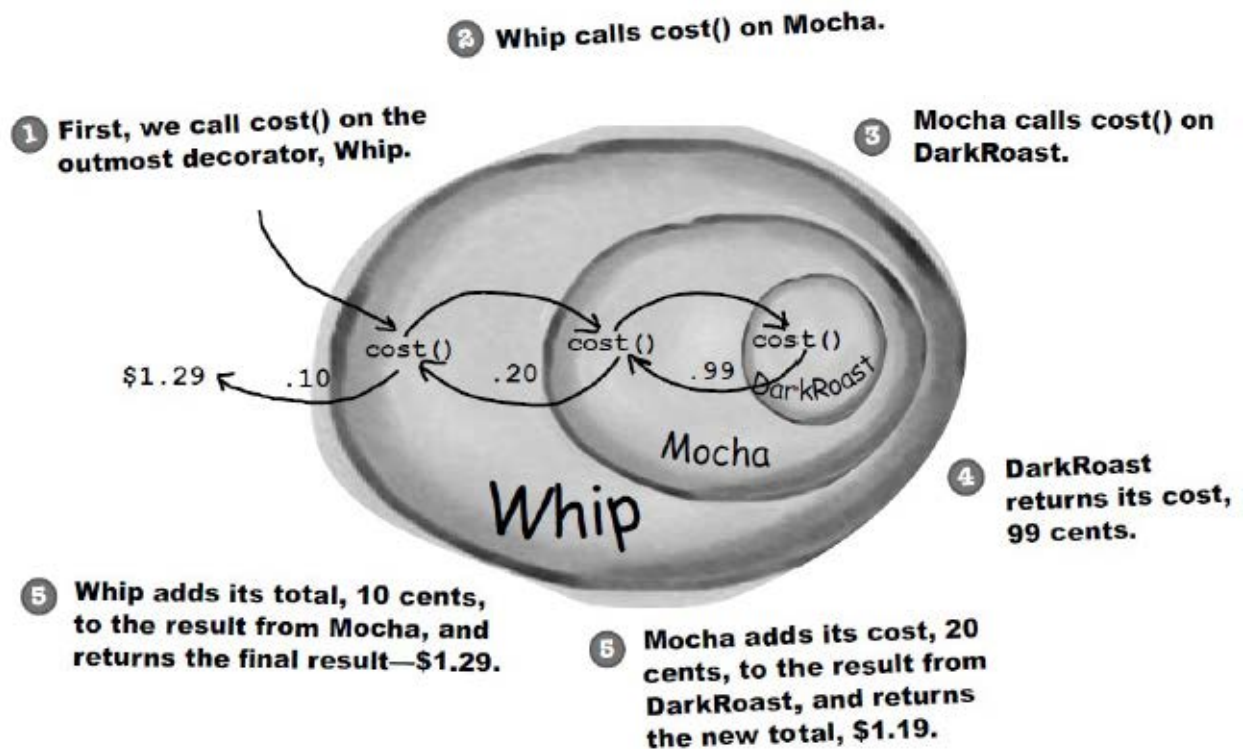
Klassen sollten offen für Erweiterungen, aber geschlossen für Änderungen sein.

Das Ziel ist es, die leichte Erweiterung der Klassen zu integrierten neuen Verhalten, ohne den bestehenden Code zu verändern, zu ermöglichen.

Eine Erweiterung im Sinne des Open-Closed-Prinzips ist beispielsweise die Vererbung. Diese verändert das vorhandene Verhalten der Einheit nicht, erweitert aber die Einheit um zusätzliche Funktionen oder Daten. Überschriebene Methoden verändern auch nicht das Verhalten der Basisklasse, sondern nur das der abgeleiteten Klasse.

- Erweitere bestehende Klassen mit jedem neuen Verhalten, welches dir gefällt.
- Schreibe deine eigenen Erweiterungen!
- Es wurde sehr viel Zeit investiert, den bestehenden Code richtig und fehlerfrei zu machen: Finger weg!
- Code muss für Änderungen geschlossen bleiben.
- Es gibt Techniken, Code zu erweitern, ohne ihn direkt zu modifizieren

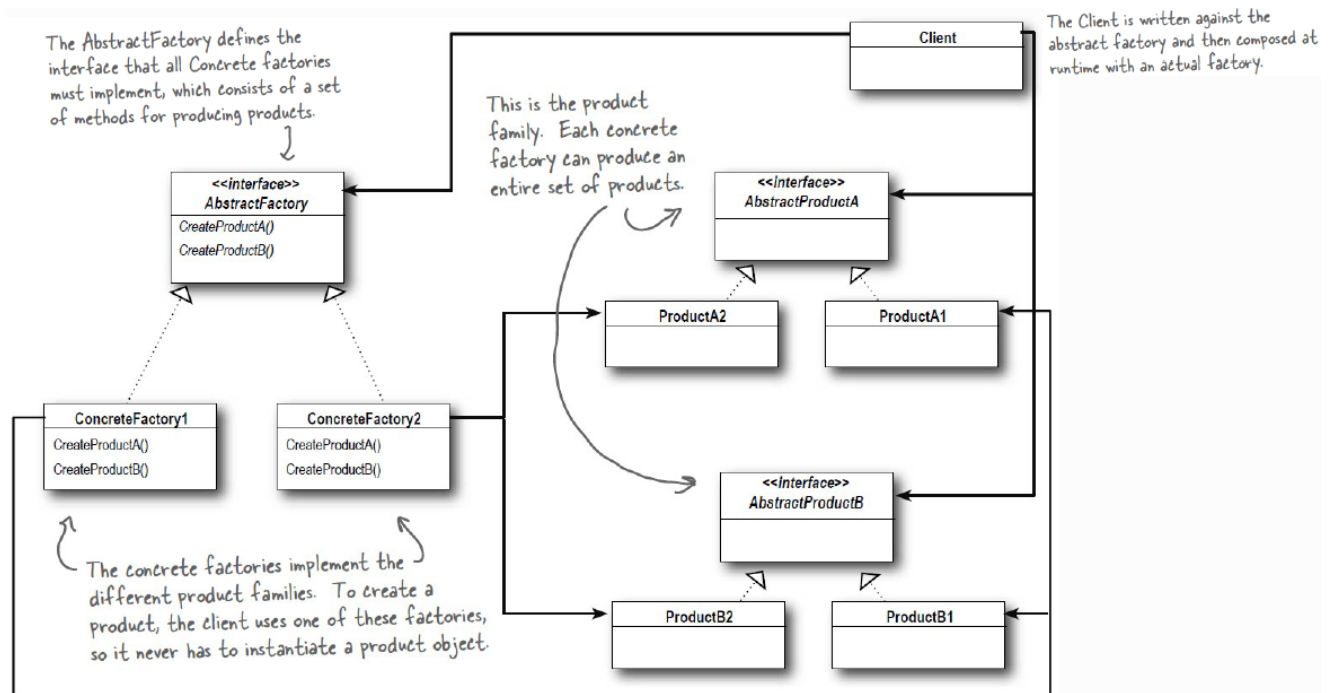
- Das Offen-Geschlossen Prinzip ÜBERALL anzuwenden ist:
 - Verschwendung
 - unnötig
 - und kann zu komplexen, schwer verständlichen Code führen!



Factory Pattern

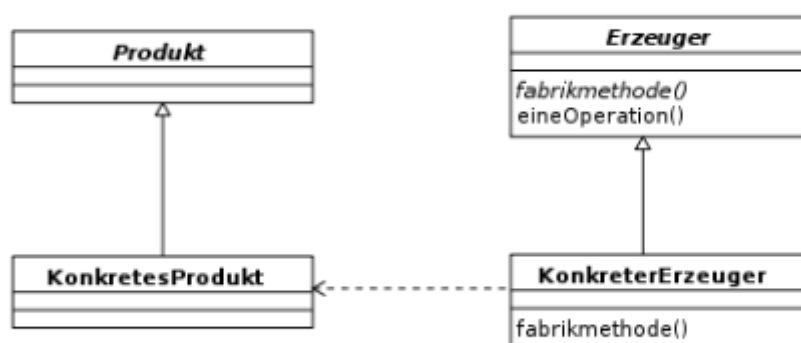
- Alle Factories kapseln die Objekt-Erstellung
- Die einfache Fabrik (Simple Factory) ist eine unkomplizierte Möglichkeit, Clients von konkreten Klassen zu entkoppeln, ist aber kein echtes Entwurfsmuster
- Factory-Method stützt sich auf Vererbung: Die Objekterstellung wird an Unterklassen delegiert, die die Fabrikmethode implementieren, um Objekte zu erstellen
- Der Zweck von Factory-Method ist es, einer Klasse zu ermöglichen, die Instanziierung bis in ihre Unterklassen zu verzögern
- Das Prinzip der Umkehrung der Abhängigkeiten leitet uns an, Abhängigkeiten von konkreten Typen zu vermeiden und Abstraktionen anzustreben.

- Die Abstract Factory bietet eine Schnittstelle zum Erstellen von Familien verwandter oder zusammenhängender Objekte an, ohne konkrete Klassen anzugeben:



- Link für ein Java-Codebeispiel (Factory Method): <http://www.fluffycat.com/Java-Design-Patterns/Factory-Method/>
- Link für ein Java-Codebeispiel (Abstract Factory): <http://www.fluffycat.com/Java-Design-Patterns/Abstract-Factory/>

UML-Klassendiagramm

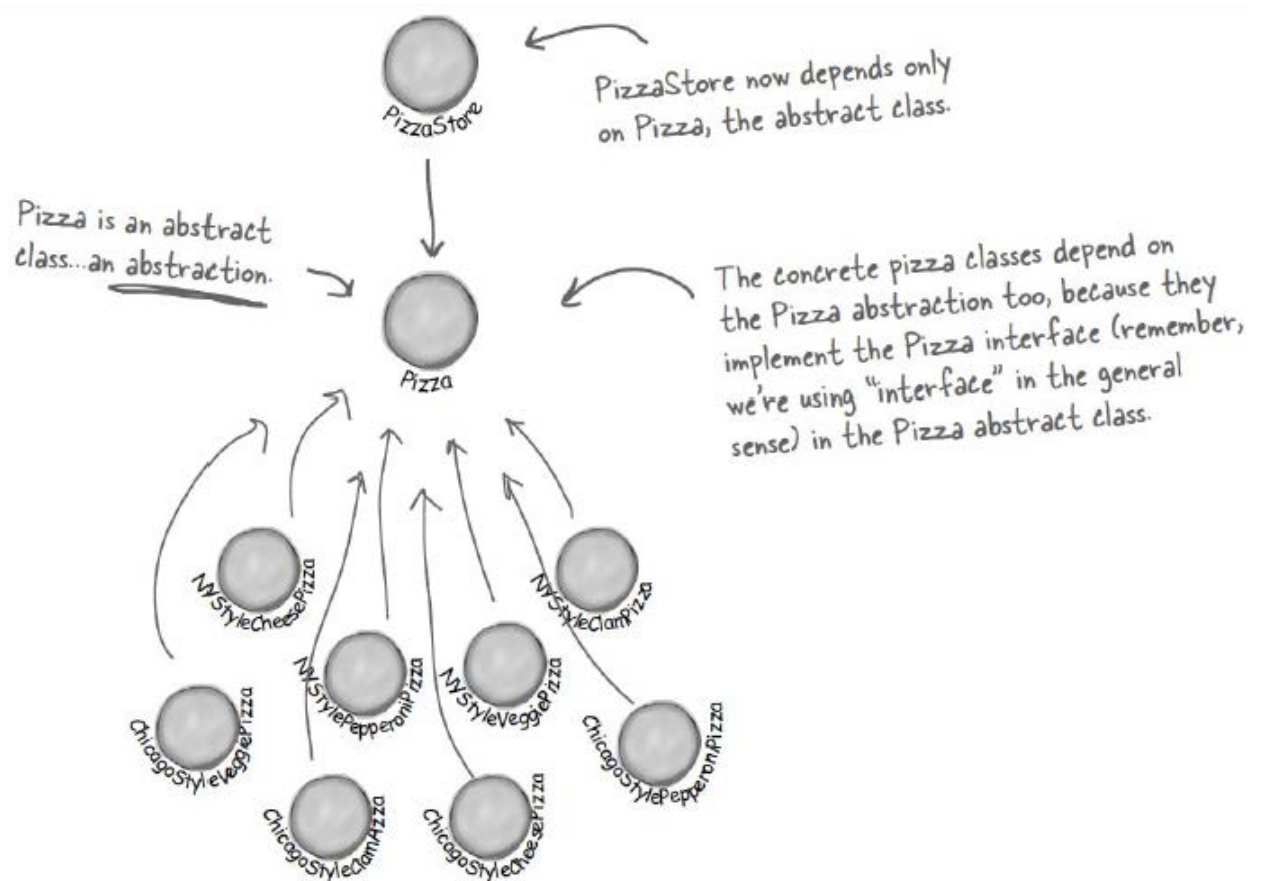


Design Prinzipien

Prinzip

Prinzip der „Umkehrung der Abhängigkeiten“ - Hänge von Abstraktionen ab und nicht von konkreten Klassen.

- **Ziel:** Abhängigkeiten von konkreten Klassen zu reduzieren
- Hochstufige Komponenten sollen nicht von niederstufigen Elementen abhängig sein, stattdessen sollten sich beide auf Abstraktionen stützen!
- **Anwendung des Prinzips:**



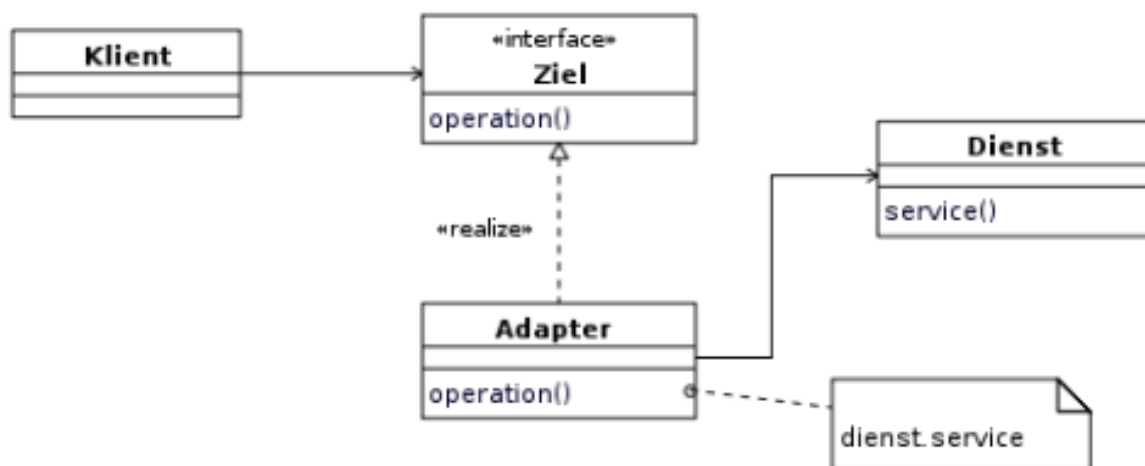
- **Richtlinien für das Prinzip:**
 - Keine Variable sollte eine Referenz auf eine konkrete Klasse halten
 - Keine Klasse sollte von einer konkreten Klasse abgeleitet sein
 - Keine Methode sollte eine implementierte Methode einer ihrer Basisklassen überschreiben

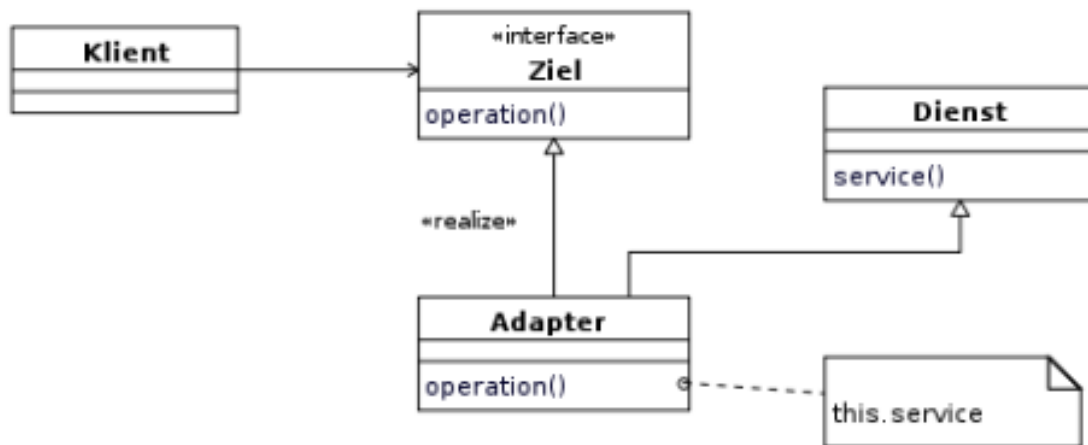
Adapter Pattern

- Das Adapter Pattern dient zur Übersetzung einer Schnittstelle in eine andere. Dadurch wird die Kommunikation von Klassen mit zueinander inkompatiblen Schnittstellen ermöglicht.
- **Vorteile:**
 - Die Vorteile eines Klassenadapters bestehen darin, dass er sich genau einer Zielklasse anpasst und dadurch nur das Verhalten der Zielklasse überschreiben kann.
 - Der Objektadapter kann auch Unterklassen mit anpassen.
- **Nachteile:**
 - Nachteilig wirkt sich aus, dass ein Klassenadapter nicht zur automatischen Anpassung von Unterklassen verwendet werden kann.
- **Link für ein Java-Codebeispiel:** <http://www.fluffycat.com/Java-Design-Patterns/Adapter/>

UML-Klassendiagramm

Adapter mit Delegation (Objektadapter)



Adapter mit Vererbung (Klassenadapter)**Design Prinzipien****Prinzip 1**

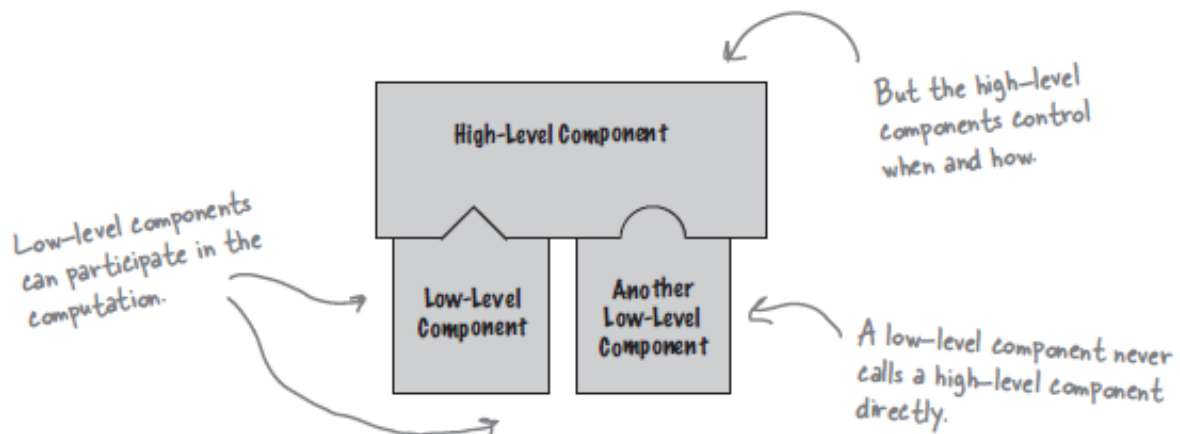
Prinzip des geringsten Wissens – Spreche nur zu deinen unmittelbaren Freunden.

- Dieses Prinzip führt zur Reduzierung der Interaktionen zwischen Objekten zu ein paar engen Freunden.
- Beim Entwerfen eines Systems, für jedes Objekt, sei vorsichtig, wegen der Anzahl der Klassen, mit denen es interagiert und auch wie es mit diesen Klassen zum Interagieren kommt.
- Dieses Prinzip verhindert uns von der Erstellung eines Designs, das eine große Anzahl von miteinander gekoppelten Klassen hat, so dass Änderungen in einem Teil des Systems zu anderen Teilen kaskadieren. Wenn Sie eine Menge von Abhängigkeiten zwischen vielen Klassen bauen, bauen Sie ein fragiles System, welches teuer in der Unterhaltung und komplex für andere zu verstehen, sein wird.

Prinzip 2

Hollywood Prinzip – Rufe uns nicht an, wir werden dich anrufen.

- Das Hollywood-Prinzip gibt uns eine Möglichkeit der Verhinderung der „Abhängigkeit rot“. Abhängigkeit rot geschieht, wenn folgendes zutrifft: Hoch-Level Komponente abhängig von Nieder-Level Komponenten abhängig von Hoch-Level Komponenten abhängig von Seitwärtskomponenten abhängig von Nieder-Level-Komponenten, und so weiter.
- Wenn rot eingesetzt wird, niemand kann das Design des Systems leicht verstehen.
- Mit dem Hollywood-Prinzip ermöglichen wir den Nieder-Level Komponenten, dass sie sich in ein System anhängen, aber die Hoch-Level Komponente bestimmen, wann diese gebraucht werden und wie.
- Mit anderen Worten: Die Hoch-Level Komponente geben den Nieder-Level Komponenten ein „Don't call us, we'll call you“ Abhandlung.



Iterator und Composite Pattern

▪ Iterator Pattern:

- **Vorteile:**

- ◆ Die Implementierung der zu Grunde liegenden Datenstruktur bleibt verborgen.

- **Nachteile:**

- ◆ Bei polymorphen Iteratoren muss man den Preis für virtuelle Methoden zahlen.
- ◆ Wenn der Iterator sein Aggregat kennt und/oder das Aggregat über seine Iteratoren Buch führt, verteuern sich vor allem das Erzeugen und Vernichten von Aggregaten.

- **Link für ein Java-Codebeispiel:** <http://www.fluffycat.com/Java-Design-Patterns/Iterator/>

▪ Composite Pattern:

- **Vorteile:**

- ◆ Einheitliche Behandlung von Primitiven und Kompositionen
- ◆ Leichte Erweiterbarkeit um neue Blatt- oder Container-Klassen

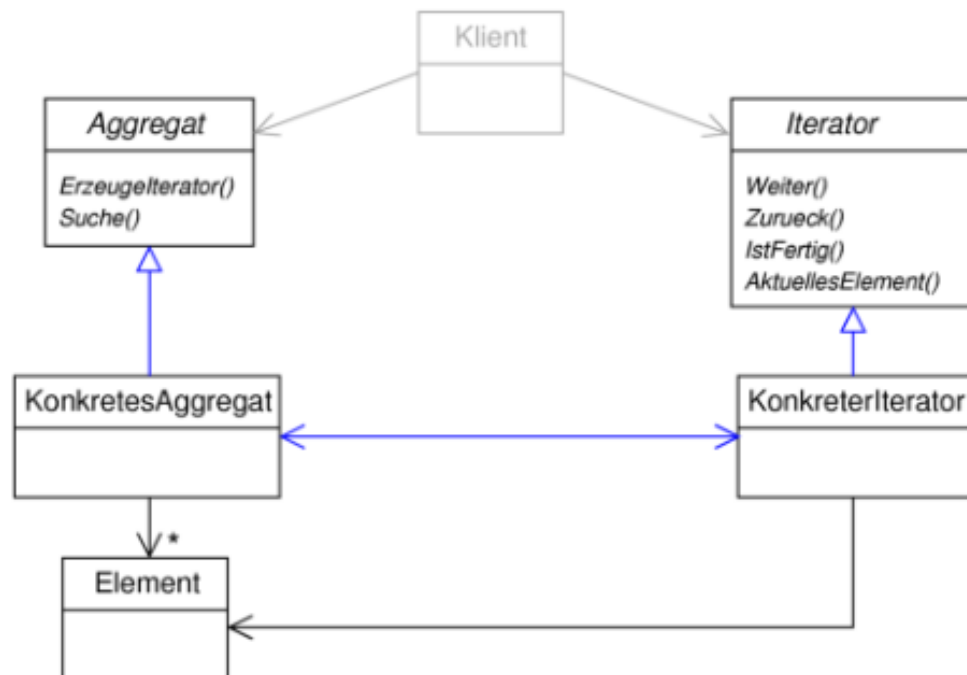
- **Nachteile:**

- ◆ Ein zu allgemeiner Entwurf erschwert es, Kompositionen auf bestimmte Klassen zu beschränken. Das Typsystem der Programmiersprache bietet dann keine Hilfe mehr, so dass Typprüfungen zur Laufzeit nötig werden.

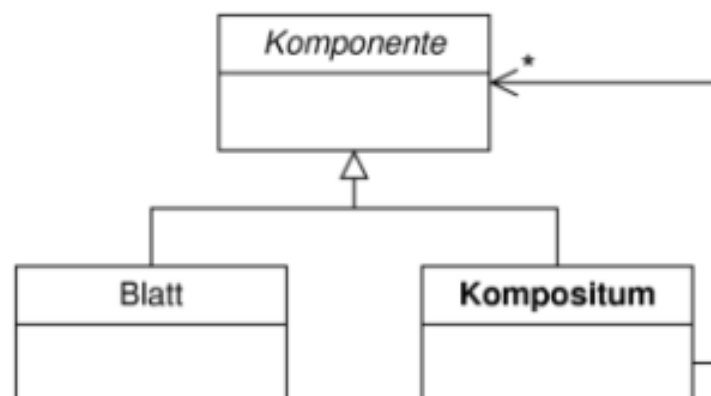
- **Link für ein Java-Codebeispiel:** <http://www.fluffycat.com/Java-Design-Patterns/Composite/>

UML-Klassendiagramm

Iterator Pattern



Composite Pattern



Design Prinzipien

Prinzip

Einzelverantwortung – Es sollte nie mehr als ein Grund geben, eine Klasse zu ändern.

- Jede Klasse hat nur eine fest definierte Aufgabe zu erfüllen.
- In einer Klasse sollten lediglich Funktionen vorhanden sein, die direkt zur Erfüllung dieser Aufgaben beitragen.
- Die Modularisierung erfolgt nicht nach den realweltlichen Entitäten, sondern auf der Basis von Änderungserwartungen.
- **Vorteile:**
 - Liefert eine Operationalisierung des ansonsten nur schwer fassbaren Prinzips der hohen Kohäsion.
- **Nachteile:**
 - Die Quelle künftiger Änderungen ist oftmals schwer vorsehbar. Das Prinzip lässt sich nur in solchen Fällen anwenden, in denen ausnahmsweise in dieser Hinsicht Klarheit besteht.
 - Eine übertriebene Anwendung des Prinzips könnte zu übermäßiger Modularisierung führen.
 - Andere Aspekte hoher Kohäsion werden ausgeblendet.

Quellenangaben

-
- [1] Autor: Eric Freeman, Elisabeth Freeman, Kathy Sierra, Bert Bates;
Titel: Herauskopierte Design Prinzipien aus "Head First Design Patterns";
Verlag: First Edition; 2004; O'Reilly Media";
Quelle: <http://elearning.tgm.ac.at/mod/resource/view.php?id=20001>;
zuletzt abgerufen am: 14.12.2013
- [2] Autor: Wikipedia;
Titel: Decorator;
Quelle: <http://de.wikipedia.org/wiki/Decorator>;
zuletzt geändert am: 02.08.2013
zuletzt abgerufen am: 14.12.2013
- [3] Autor: Wikipedia;
Titel: Beobachter (Entwurfsmuster)
Quelle: [http://de.wikipedia.org/wiki/Beobachter_\(Entwurfsmuster\)#UML-Diagramm](http://de.wikipedia.org/wiki/Beobachter_(Entwurfsmuster)#UML-Diagramm);
zuletzt geändert am: 08.08.2013
zuletzt abgerufen am: 14.12.2013
- [4] Autor: Wikipedia;
Titel: Strategie (Entwurfsmuster);
Quelle: [http://de.wikipedia.org/wiki/Strategie_\(Entwurfsmuster\)](http://de.wikipedia.org/wiki/Strategie_(Entwurfsmuster));
zuletzt geändert am: 25.05.2013
zuletzt abgerufen am: 14.12.2013
- [5] Autor: Wikipedia;
Titel: Adapter (Entwurfsmuster);
Quelle: [http://de.wikipedia.org/wiki/Adapter_\(Entwurfsmuster\)](http://de.wikipedia.org/wiki/Adapter_(Entwurfsmuster));
zuletzt geändert am: 18.10.2013
zuletzt abgerufen am: 14.12.2013
- [6] Autor: Wikipedia;
Titel: Iterator (Entwurfsmuster);
Quelle: [http://de.wikipedia.org/wiki/Iterator_\(Entwurfsmuster\)](http://de.wikipedia.org/wiki/Iterator_(Entwurfsmuster));
zuletzt geändert am: 22.08.2013
zuletzt abgerufen am: 14.12.2013
- [7] Autor: Wikipedia;
Titel: Kompositum (Entwurfsmuster);
Quelle: [http://de.wikipedia.org/wiki/Kompositum_\(Entwurfsmuster\)](http://de.wikipedia.org/wiki/Kompositum_(Entwurfsmuster));
zuletzt geändert am: 12.11.2013
zuletzt abgerufen am: 14.12.2013
- [8] Autor: Wikipedia;
Titel: Fabrikmethode;
Quelle: <http://de.wikipedia.org/wiki/Fabrikmethode>;
zuletzt geändert am: 04.09.2013
zuletzt abgerufen am: 14.12.2013
- [9] Autor: Matthias Kleine;
Titel: Prinzipien der Softwareentwicklung;
Quelle: <http://prinzipien-der-softwaretechnik.blogspot.co.at/2013/01/das-single-responsibility-prinzip.html>;
zuletzt geändert am: 11.01.2013
zuletzt abgerufen am: 14.12.2013
-

- [10]** Autor: Larry Truett;
Titel: Java Design Patterns Abstract Factory;
Quelle: <http://www.fluffycat.com/Java-Design-Patterns/Abstract-Factory/>;
zuletzt abgerufen am: 14.12.2013
- [11]** Autor: Larry Truett;
Titel: Java Design Patterns Factory Method;
Quelle: <http://www.fluffycat.com/Java-Design-Patterns/Factory-Method/>;
zuletzt abgerufen am: 14.12.2013
- [12]** Autor: Larry Truett;
Titel: Java Design Patterns Adapter;
Quelle: <http://www.fluffycat.com/Java-Design-Patterns/Adapter/>;
zuletzt abgerufen am: 14.12.2013
- [13]** Autor: Larry Truett;
Titel: Java Design Patterns Composite;
Quelle: <http://www.fluffycat.com/Java-Design-Patterns/Composite/>;
zuletzt abgerufen am: 14.12.2013
- [14]** Autor: Larry Truett;
Titel: Java Design Patterns Decorator;
Quelle: <http://www.fluffycat.com/Java-Design-Patterns/Decorator/>;
zuletzt abgerufen am: 14.12.2013
- [15]** Autor: Larry Truett;
Titel: Java Design Patterns Iterator;
Quelle: <http://www.fluffycat.com/Java-Design-Patterns/Iterator/>;
zuletzt abgerufen am: 14.12.2013
- [16]** Autor: Larry Truett;
Titel: Java Design Patterns Observer;
Quelle: <http://www.fluffycat.com/Java-Design-Patterns/Observer/>;
zuletzt abgerufen am: 14.12.2013
- [17]** Autor: Larry Truett;
Titel: Java Design Patterns Strategy;
Quelle: <http://www.fluffycat.com/Java-Design-Patterns/Strategy/>;
zuletzt abgerufen am: 14.12.2013
-