

# Entwurfsmuster - von Kopf bis Fuß

## Zusammenfassung

### Inhaltsverzeichnis

<b>1 Entwurfsprinzipien</b>	<b>3</b>
<b>2 Strategy-Muster</b>	<b>4</b>
2.1 Problemstellung . . . . .	4
2.2 Lösung . . . . .	4
2.3 Erklärung des Musters . . . . .	4
<b>3 Decorator-Muster</b>	<b>6</b>
3.1 Problemstellung . . . . .	6
3.2 Lösung . . . . .	6
3.3 Erklärung des Musters . . . . .	6
<b>4 Observer-Muster</b>	<b>9</b>
4.1 Problemstellung . . . . .	9
4.2 Lösung . . . . .	9
4.3 Erklärung des Musters . . . . .	10
4.4 Aufgabe zu den Entwurfsprinzipien . . . . .	11
<b>5 Singleton-Muster</b>	<b>13</b>
5.1 Definition . . . . .	13
5.2 Erklärung . . . . .	13
<b>6 Factory-Muster</b>	<b>14</b>
6.1 Problemstellung . . . . .	14
6.2 Lösung . . . . .	14
6.3 Problemstellung: Erweitert . . . . .	14
6.4 Lösung . . . . .	14
<b>7 Command-Muster</b>	<b>16</b>
7.1 Problemstellung . . . . .	16

<b>8</b>	<b>Visitor-Muster</b>	<b>17</b>
8.1	Problemstellung . . . . .	17
8.2	Erklärung des Musters . . . . .	17
8.3	Vorteile . . . . .	17
8.4	Nachteile . . . . .	17

## 1 Entwurfsprinzipien

Essenz jedes Musters kurz zusammengefasst.

- Entwurfsprinzipien:
- Identifizieren Sie die Aspekte Ihrer Anwendung, die sich aendern koennen und trennen Sie sie von denen, die konstant bleiben.
- Programmieren Sie auf eine Schnittstelle, nicht auf eine Implementierung.
- Ziehen Sie die Komposition der Vererbung vor.
- Streben Sie bei Entwuerfen mit interagierenden Objekten nach lockerer Kopplung (einfachere Erweiterung moeglich).
- Klassen sollten fuer Erweiterung offen, aber fuer Veraenderung geschlossen sein. (Mit Vorsicht zu betrachten, dies fuehrt zu hoeherer Komplexitaet und sollte nur in sinnvollen Faellen angewendet werden.)
- Stuetzen Sie sich auf Abstraktionen. Stuetzen Sie sich nicht auf konkrete Klassen.

## 2 Strategy-Muster

### 2.1 Problemstellung

- Es soll eine Simulation fuer ein Spiel mit diversen Entenarten geschrieben werden.
- Entenarten besitzen verschiedene Eigenschaften.
- In manchen Faellen nicht nur verschiedene Auspraegungen, sondern sogar nicht existent.
- Verschiedene Arten sollen aber alle von der selben Klasse erben.

### 2.2 Lösung

Problem hierbei (Lsg.): Wenn dies direkt in die Klasse geschrieben wird ist diese sehr schlecht wartbar, denn die Klassen selbst sind sehr schlecht wartbar. Die Loesung hierbei, das Strategy-Muster. Man versucht aehnliche Algrothmen wie z.B. das Flug- oder Quakverhalten in eine Gruppe/Familie von Algorithmen zusammenzufassen (zu kapseln). Dies geschieht mithilfe verschiedener Interfaces z.B. Quakverhalten / Flugverhalten. Die Mutterklasse besitzt entsprechende Felder von den Typen Quak-/Flugverhalten. Diese werden entsprechend gesetzt und in den jeweiligen Methoden aufgerufen. Die aufrufenden erbbenden Klassen implementieren das entsprechende Interface und koennen so das gewuenschte Verhalten implementieren. Hierbei gestaltet sich die Wartung des Codes deutlich einfacher.

### 2.3 Erklärung des Musters

**Definition** Das Strategy-Muster definiert eine Familie von Algorithmen, kapselt sie einzeln und macht sie austauschbar. Das Strategy-Muster ermoeeglicht es, den Algrithmus unabhaengig von den Clients die ihn einsetzen, variieren zu lassen.

**Vorteile** Bei diesem Entwurft koennen andere Typen von Objekten unsere Flug- und Quakverhalten wieder verwenden (Bezug zu Kap. 1), weil diese Verhalten nicht mehr in unseren Ente-Klassen verborgen sind. Wir koennen neue Verhalten hinzufuegen, ohne irgendeine unserer bestehenden Verhaltensklassen zu aendern oder Hand an eine der Enten-Klassen zu legen, die Flugverhalten nutzen.

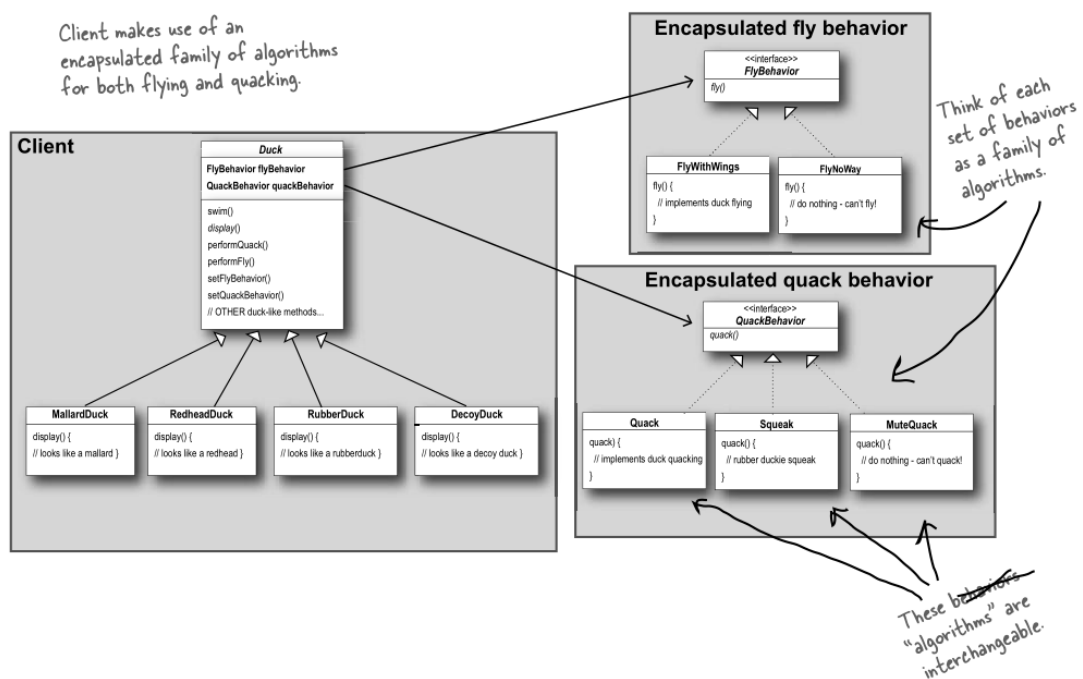


Abbildung 1: UML-Darstellung des Strategy-Musters

## 3 Decorator-Muster

### 3.1 Problemstellung

- Ein Bäcker möchte verschiedene Kaffeesorten verkaufen.
- Es soll möglich sein verschiedene Bonuszutaten (z.B. Sahne, Zucker, etc.) hinzuzufügen.
- Hierbei kann die Menge der variieren. Es ist also möglich sowohl eine Extra-Portion, als auch zwei extra Portionen Sahne dazu zu bestellen.
- Ausserdem ist es möglich verschiedene Zutaten zu kombinieren.
- Problem: Wenn man sich auf einfache Vererbung beschränkt wird die Struktur sehr schnell unübersichtlich.

### 3.2 Lösung

Es wird zu allererst eine abstrakte Klasse Getränk gebildet, von der sowohl die Zutaten als auch die Kaffeesorten erben. Die Kaffeesorten tun dies direkt. Sie besitzen zwei Felder für die Beschreibung und den Preis, über entsprechende Getter sind sie abrufbar. Nun bildet man eine abstrakte Wrapper-Klasse von der sämtliche Zutaten erben sollen. Jede Zutat erbt von der Wrapper Klasse und müssen die Getter für die Beschreibung / Preis neu implementieren, denn jede Zutat hält eine Referenz auf ein Getränk welches im Konstruktor gesetzt wird. Bei den Gettern wird die Beschreibung / Preis an die Werte des Getränks angehängt bzw. drauf addiert. Da alle Klassen von der selben Superklasse ableiten kann die Referenz nicht nur einen Kaffee ohne Zutaten sondern auch einen mit diesen beinhalten. Die Wrapper-Klasse ist letztendlich dazu da eine Schnittstelle herzustellen, damit die es sowohl geordnet ist und die Referenz wie erwähnt bereits modifizierte Objekte halten kann.

Beim Bestellen wird eine zugrunde liegende Kaffeesorte gewählt (z.B. Espresso). Diese wird mit den einzelnen Zutaten "dekoriert" (Bsp. siehe main Klasse im Code).

### 3.3 Erklärung des Musters

**Definition** Decorator - Fügt einem Objekt dynamisch zusätzliche Verantwortlichkeit hinzu. Dekorierer bieten eine flexible Alternative zur Ableitung von Unterklassen zum Zweck der Erweiterung der Funktionalität.

#### Punkt für Punkt (S.105)

- Vererbung ist eine Form von Erweiterung, aber nicht notwendigerweise der beste Weg, um Ihren Entwurf Flexibilität zu verleihen.
- Unsere Entwürfe sollen die Erweiterung von Verhalten ermöglichen, ohne dass dazu bestehender Code geändert werden müsste.

- Oft koennen Komposition und Delegierung verwendet werden, um zur Laufzeit neue Verhalten hinzuzufuegen.
- Fuer die Erweiterung von Verhalten bietet das Decorator-Muster eine Alternative zur Ableitung von Unterklassen.
- Das Decorator-Muster schliesst einen Satz von Dekorierer-Klassen ein, die verwendet werden, um konkrete Komponenten einzupacken.
- Dekorierer-Klassen spiegeln den Typ der Komponente wider, die sie dekorieren. (Sie haben sogar tatsaechlich den gleichen Typ wie die Komponente, die sie dekorieren, entweder durch Vererbung oder durch die Implementierung eines Interface.)
- Dekorierer aendern das Verhalten der Komponenten, indem sie vor und / oder nach (oder auch an Stellen von) Methodenaufrufen auf der Komponente neue Funktionalitaeten hinzufuegen.
- Sie koennen eine Komponente mit einer beliebig Zahl von Dekorierern einpacken.
- Dekorierer sind fuer die Clients der Komponente ueblicherweise transparent, ausser wenn sich der Client auf den konkreten Typ der Komponente stuetzt.
- Dekorierer koennen in Ihren Entwuerfen zu vielen kleinen Objekten fuerhen, und eine uebermaessige Verwendung kann den unuebersichtlich machen.

#### **Wann das Decorator-Muster ungeeignet ist**

- Wenn Sie mit Code arbeiten, der auf den Typ einer konkreten Komponente angewiesen ist, zerbrechen Dekorierer diesen Code. Solange Sie nur Code auf Basis des abstrakten Komponententyps schreiben, bleibt die Verwendung von Dekorierern fuer ihren Code transparent. (Muster z.B. fuer Rabatsystem beim Kaffeehaus aus diesem Kapitel ungeeignet.)
- Dekorierer sollen den Objekten, die sie einpacken Verhalten hinzufuegen. Wenn Sie beginnen, auf mehreren Schichten in der Dekoriererkette zu blicken, dann strecken Sie Decorator ueber seinen eigentlichen Zweck.

#### **Gut zu wissen**

- Die java.io-Klassen verwenden ebenfalls das Decorator-Muster. Es gibt diverse Streams mit jeweils unterschiedlichen Aufgaben, und es gibt die Klasse `FilterInputStream` die als Superklasse fuer verschiedene Dekorierer arbeitet.

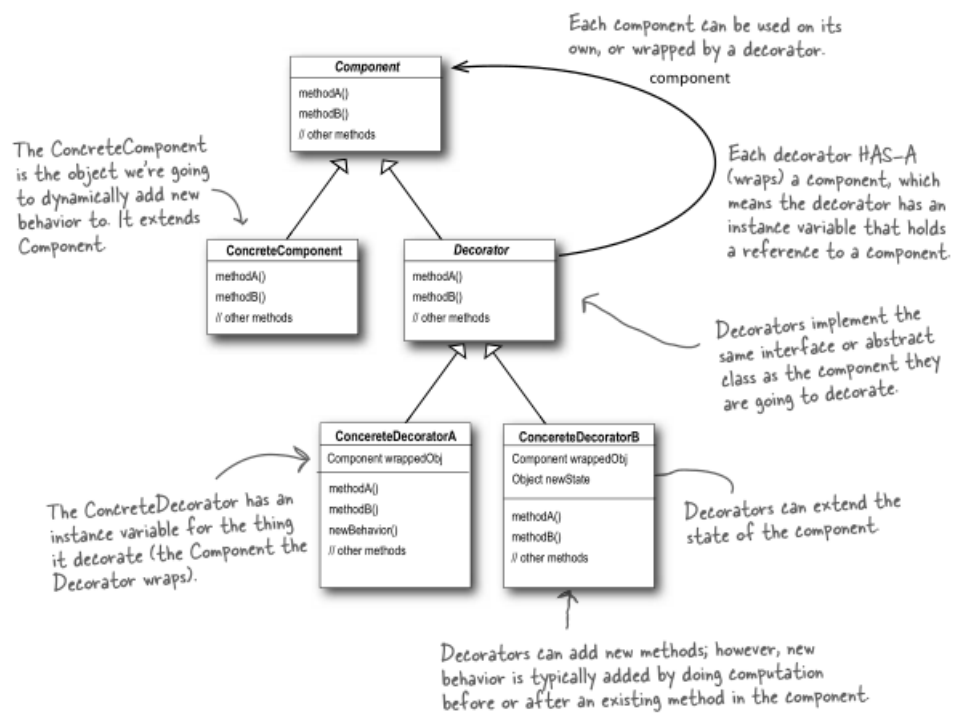


Abbildung 2: UML-Darstellung des Decorator-Musters



## 4 Observer-Muster

### 4.1 Problemstellung

- Wetterstation liefert Daten (Luftdruck, Temperatur, etc.). - Gefragt ist ein WetterDaten-Objekt. - Diverse Anzeigegeraete sollen in der Lage sein, sich die Daten vom WetterDaten-Objekt zu ziehen und entsprechend darzustellen. Problematisch wird es, wenn verschiedene Anzeigegeraete nur bestimmte Daten anzeigen sollen. - Besonderes Augenmerk bei dieser Aufgabe, die Daten sollen mit einem einzigen Aufruf aktualisiert werden. - Wichtig: Es soll moeglich sein neue Anzeigegeraete einzubinden. Hierbei soll der zus. Aufwand so gering wie moeglich gehalten werden.

### 4.2 Lösung

Observer-Muster (Grundsatzliches Prinzip): Man definiert ein Interface z.B. `SSubjekt` das die Schnittstelle fuer das WetterDaten-Objekt angibt. Hierzu gehoeren die Methoden um einen Beobachter zu registrieren, zu entfernen oder diesen ueber die Aktualisierung der Daten zu informieren (bzw. ihm diese zukommen zu lassen). Die Idee beim Observer-Muster ist naemlich, dass ein Datenobjekt mit entsprechenden Feldern ein Attribut mit einer Liste von Beobachtern haelt die Zugriff auf die Felder besitzen. Dies kann mithilfe einer Uebergabe des Objektes geschehen oder aber mit einer tatsaechlichen Uebergabe von Parametern. Mit den Methoden zum Registrieren und Entfernen werden die Beobachter der Liste hinzugefuegt oder entfernt. Das WetterDaten-Objekt aus dem Bsp. implementiert jetzt dieses Interface. Anschliessend wird ein zweites Interface fuer die Beobachter (im Bsp. die verschiedenen Anzeige-Klassen) geschrieben. Die Beobachter informieren dieses jeweils und haben damit Zugriff auf eine Methode zum Aktualisieren der Werte. Hier koennen direkt die Parameter uebergeben werden, oder aber das Datenobjekt selbst (um mit Gettern die Daten zu uebermitteln).

Observable-Muster aus `java.util.Observable` / `Observer` Es gibt bereits ein vorgefertigtes Observer-Muster in der Standardbibliothek `util`. Hier erbt das jeweilige Datenobjekt von der Superklasse `Observable`. Diese Klasse bringt bereits die Liste aus Beobachtern und weitere Methoden wie die `setChanged()` und `notifyObservers()` Methoden. Wichtig hierbei ist zu beachten, dass man erst die Beobachter benachrichtigen kann wenn `setChanged` aktiviert wurde. `NotifyObservers()` ist ausserdem ueberladen, denn es ist moeglich neben dem eigentlichen Datenobjekt (hier `WetterDaten-Objekt`) noch ein anderes zu uebergeben, welches z.B. weitere Werte enthalten kann. Zum genauen Aufbau siehe Code.

### Nachteile der Standardbibliothek

- Reihenfolge der Auswertung aendert sich -> Ungeeignet fuer Anwendungen wo dies von Bedeutung sein sollte.
- `Observable` ist eine Klasse und kein Interface -> Klasse kann daher keine andere Klasse erweitern und schlecht wartbar sowie wiederverwendbar.

- Observable schuetzt entscheidende Methoden -> z.B. `setChanged()`, kann daher auch nur von erbenden Klassen aufgerufen werden.

Vorteile: Siehe `AufgZuDenEntwurfsprinzipien.txt`

### 4.3 Erklärung des Musters

**Definition** Das Observer-Muster definiert eine Eins-zu-viele-Abhaengigkeit zwischen Objekten in der Art, dass alle abhaengigen Objekte benachrichtigt werden, wenn sich der Zustand des einen Objekts aendert.

#### Gutes Alltagsbeispiel Zeitungsabonnementsdienst

- Man moechte Abonnent werden -> man wird auf die Liste der Abonnennten gesetzt.
- Neue Ausgabe wird ausgegeben (aktualisiert) -> Alle Abonnemenen der Liste werden benachrichtigt.
- Bei der Standardbibliothek `utils` ist bereits ein solches Muster vorgegeben mit dem es fuer die Abonnennten sogar moeglich ist sich jederzeit per Getter-Methoden die Daten zu ziehen ohne das eine entsprechende Methode im Subjekt aufgerufen werden muss.
- Auf Wunsch ist es ebenfalls moeglich wieder aus der Liste der Abonnennten auszutreten.

#### Punkt fuer Punkt Zusammenfassung (S. 74)

- Das Observer-Musster definiert ein Eins-zu-viele-Verhaeltnis zwischen Objekten.
- Subjekte oder, wie wir sie auch kennen, Observables aktualisieren Beobachter ueber eine Schnittstelle.
- Die Beobachter sind insofern locker angebunden, als das Bbservable ueber sie nichts anderes weiss, als dass sie das Interface `Observer` implementiern.
- Sie koennen Daten aus dem Observable herausgeben oder herausziehen, wenn Sie das Muster verwenden (wobei das Herausziehen als die "richtigere" Methode betrachtet wird).
- Verlassen Sie sich nicht auf eine bestimmte Reihenfolge der Benachrichtigung Ihrer Beobachter.
- Java besitzt eine Reihe von Implementierungen des Observer-Musters, einschliesslich des allgemeinen `java.util.Observable`.
- Nehmen Sie sich vor den Haken der Implementierung von `java.util.Observable` in Acht. (Siehe Erklaerung Kap. 1).

- Haben Sie keine Hemmungen, Ihre eigene Observable-Implementierung zu schreiben, wenn dies erforderlich ist.
- Swing macht wie andere GUI-Frameworks extensiven Gebrauch vom Observer-Muster.
- Sie finden das Muster auch an vielen anderen Orten einschliesslich JavaBeans und RMI.

#### 4.4 Aufgabe zu den Entwurfsprinzipien

**Aufgabe zu den Entwurfsprinzipien** Beschreiben Sie fuer jedes Entwurfsprinzip, wie das Observer-Muster das Prinzip umsetzt.

##### Prinzipien

1. Identifizieren Sie die Aspekte Ihrer Anwendung, die sich aendern koennen und trennen Sie sie von denen, die konstant bleiben.
2. Programmieren Sie auf eine Schnittstelle, nicht auf eine Implementierung.
3. Ziehen Sie die Komposition der Vererbung vor.

##### Eigene Erklärung

1. Beobachter koennen sich aendern, werden daher zusammengefasst und ausgelagert.
2. Da die Beobachter lediglich ein Interface implementieren und keine Klasse erweitern, besteht hier eine lose Bindung d.h. man programmiert auf eine Schnittstelle.
3. Die beiden Parteien (Beobachter, Subjekt) implementieren jeweils nur ein Interface und erweitern keine Superklasse.

##### Erklaerungen aus dem Buch (S. 77)

1. Das, was beim Observer-Muster variiert, ist der Zustand des Objekts und die Anzahl sowie die Typen der Beobachter. Mit diesem Muster koennen Sie die Objekte variieren, die vom Zustand des Objekts abhaengig sind, ohne das Subjekt veraendern zu muessen. Das nennt man verausschaen handeln!
2. Subjekt und Beobachter nutzen beide Interfaces. Das Subjekt haelt Objekte nach, die das Interface Observer implementieren, waehrend die Beobachter sich registrieren und vom Subjekt-Interface benachrichtigt werden. Wie wir gesehen haben, haelt das die Dinge ordentlich und locker gebunden.
3. Das Observer-Muster nutyt Komposition um eine beliebige Anzahl von Beobachtern mit ihren Subjekten zu verbinden. Diese Beziehungen werden nicht durch irgendeine Art von Vererbungshierarchie implementiert. Nein sie werden zur Laufzeit durch Komposition eingerichtet!

### The Observer Pattern defined: the class diagram

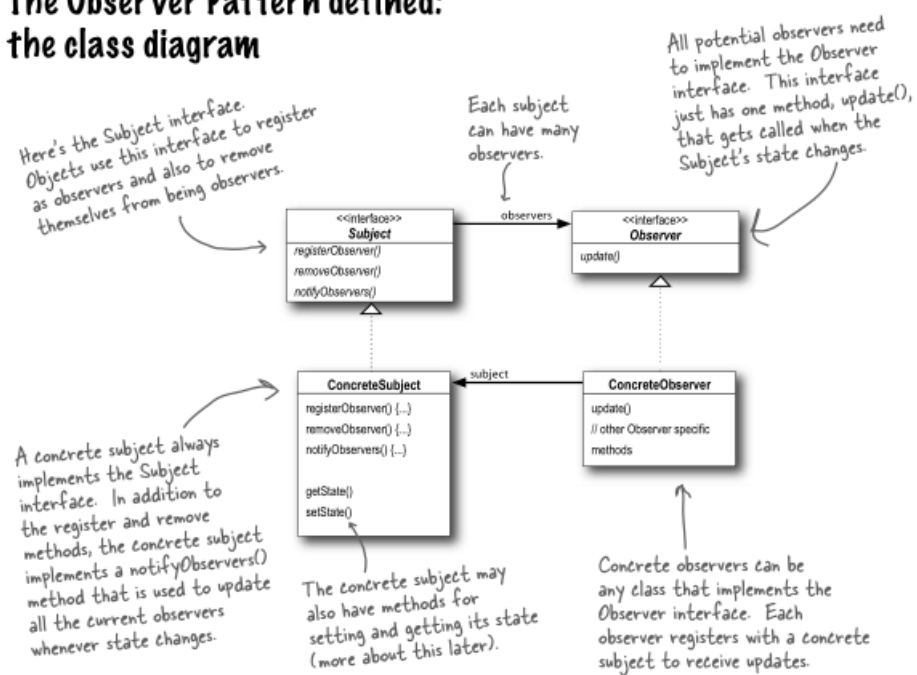
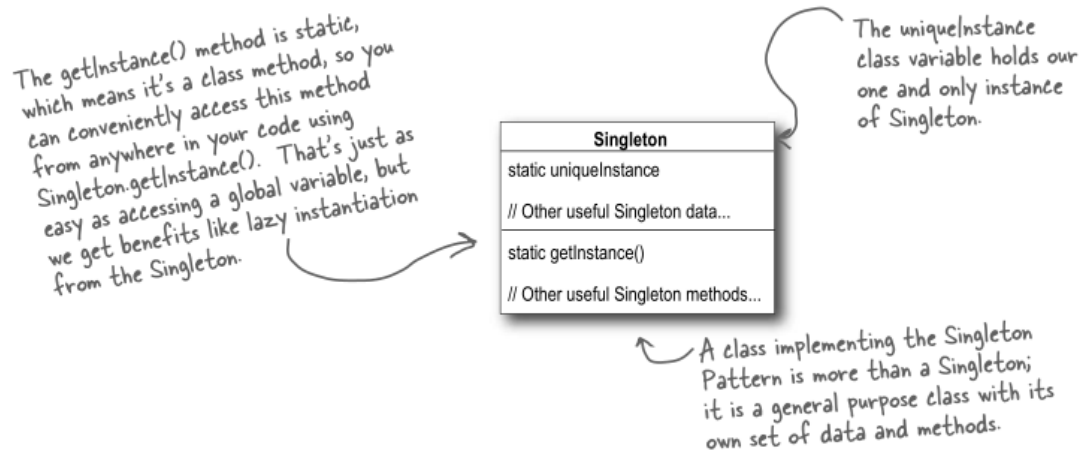


Abbildung 3: UML-Darstellung des Observer-Musters

Abbildung 4: UML-Darstellung des *einfachen* Singleton-Musters

## 5 Singleton-Muster

### 5.1 Definition

Das Singleton-Muster sicher, dass es nur eine Instanz einer Klasse gibt, und bietet einen globalen Zugriffspunkt fuer diese Instanz.

### 5.2 Erklaerung

- Wir nehmen eine Klasse und lassen sie eine einzige Instanz von sich selbst verwalten. Wir verhindern auch, dass irgendeine andere Klasse eigenstaendig eine neue Instanz erstellt. Um eine Instanz zu erhalten, muss man ueber die Klasse selbst gehen.
- Wir bieten ausserdem einen globalen Zugriffspunkt fuer die Instanz: Jedes Mal wenn Sie eine Instanz benoetigen, fragen Sie einfach be der Klasse nach, und diese reicht Ihnen, die eine Instanz. Wie Sie gesehen haben, koennen wir das so implementieren, dass das Singleton verzoeigert erstellt werden kann, was bei ressourcenintensiven Objekten besonders wichtig ist.

## 6 Factory-Muster

### 6.1 Problemstellung

Einfach

- Man moechte eine Pizzeria mit mehreren Zweigstellen erstellen.
- Bei der Zubereitung der Pizza gibt diverse Arbeitsablaeufe die von den Zweigstellen allesamt unterschiedliche abgearbeitet werden koennen (Z.B. Schneiden: Vierteln vs. in acht Stuecke schneiden, etc.).
- ‘ Problem: Um Code-Dopplung zu vermeiden sollen die Pizza in einer einzigen Klasse implementiert sein -> Es muss ein Pizza-Objekt entsprechend den Anforderungen (welche Zweigstelle?) erstellt werden.

### 6.2 Loesung

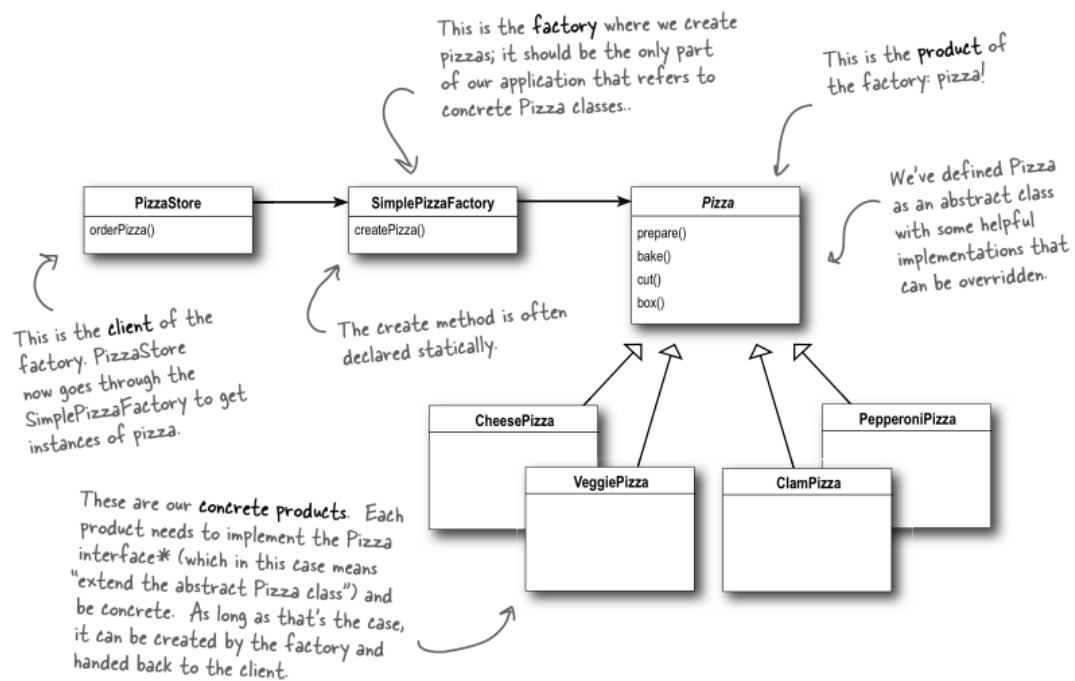
Es wird eine abstrakte Superklasse *Pizzeria* erstellt. Verschiedene Zweigstellen beerben diese. Jede dieser Zweigstellen implementiert eine Methode *erstellePizza()*. Diese dient als so genannte Factory, denn sie erstellt ein Objekt des gewünschten Typs. Bei der Erstellung wird ein entsprechender Konstruktor einer speziellen Pizza-Klasse aufgerufen. Diese Klasse implementiert die Methoden der Arbeitsschritte (backen, schneiden, etc.) entsprechend. In der Superklasse wird eine Methode deklariert ("bestellePizza()") die mit diesem Objekt arbeitet. Durch den Polymorphismus / Abstraktion ist es irrelevant mit welchem Pizza-Objekt man arbeitet, da alle erbenenden Klassen des Pizza-Typs die dort aufgerufenen Methoden implementiert / geerbt haben. Die Methode "bestellePizza()" wird auch fabrikMethode genannt.

### 6.3 Problemstellung: Erweitert

- Einzelne Zweigstellen benutzen minderwertige Zutaten.
- Wie kann man man Konsistenz bei den Zutaten sichern?

### 6.4 Loesung

noch un-  
bearbeitet

Abbildung 5: UML-Darstellung des *einfachen* Factory-Musters

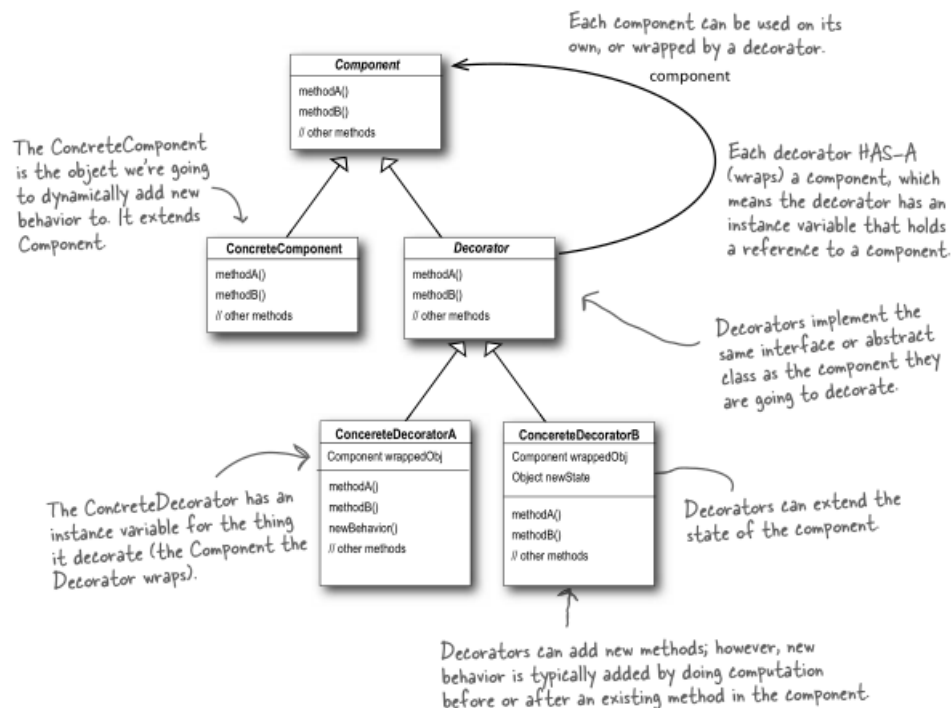


Abbildung 6: UML-Darstellung der zu implementierenden Fernsteuerung

## 7 Command-Muster

### 7.1 Problemstellung

Es soll eine Universalfernbedienung entworfen werden. Hierbei können verschiedene Haushaltsgeräte angeschlossen werden. Die Fernbedienung soll für jedes angeschlossene Gerät einen An- und Ausschalter zur Verfügung stellen. Problem hierbei. Diverse Gerätschaften besitzen haben unterschiedliche Schnittstellen. Bei manchen heißen diese schlicht und ergreifend anders als gefordert, bei anderen gibt es eine derart schlichte Kontrollfunktion nicht. Ein Beispiel wäre hier zum Beispiel ein Garagentor welches auf und zu gemacht werden kann aber auch eine Stereoanlage soll nicht nur angeschaltet werden, es soll auch ein Lied mit einer bestimmten Lautstärke gespielt werden.



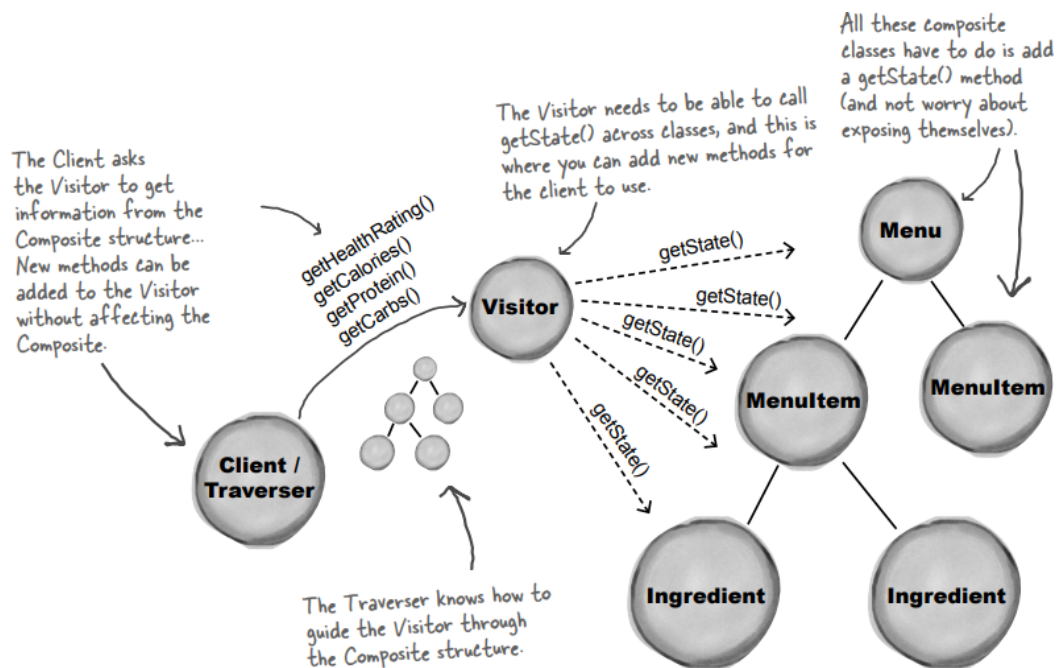


Abbildung 7: UML-Darstellung des Visitor-Musters

## 8 Visitor-Muster

Verwenden Sie das Visitor-Muster, wenn Sie ein Objekt-Kompositum um neue Fähigkeiten erweitern möchten und neue Kapselung unwichtig ist.

### 8.1 Problemstellung

Die Gäste, die das Restaurant und das Pfannkuchenhaus besuchen, sind in letzter Zeit gesundheitsbewusster geworden. Bevor sie ihr Essen bestellen, fragen sie nach Nahrungsmittelinformationen. Da beide Restaurants bereit sind Sonderbestellungen entgegenzunehmen, fragen manche Gäste sogar nach Nahrungsmittelinformationen zu einzelnen Zutaten.

### 8.2 Erklärung des Musters

Der Visitor muss jedes Element des Kompositums besuchen; diese Funktionalität steckt in einem Durchquerer-Objekt. Der Visitor wird von Durchquerer angeleitet, den Zustand von allen Objekten im Kompositum zu erfassen. Wenn der Zustand einmal erfasst ist, kann der Client den Visitor verschiedene Operationen auf dem Zustand ausführen lassen. Ist neue Funktionalität erforderlich, muss lediglich der Visitor erweitert werden.

**8.3 Vorteile**

1. Ermöglicht Operationen zu einer Kompositumsstruktur hinzuzufügen, ohne die Struktur selbst zu ändern.
2. Das Hinzufügen neuer Operationen ist relativ einfach
3. Der Code für Operationen, die der Visitor durchführt, ist zentralisiert.

**8.4 Nachteile**

1. Bei der Verwendung des Visitors wird die Kapselung der Kompositumsklassen zerstört.
2. Da die Durchquererfunktion mit in die Sache verwickelt ist, sind die Veränderungen an der Kompositumsstruktur schwieriger.