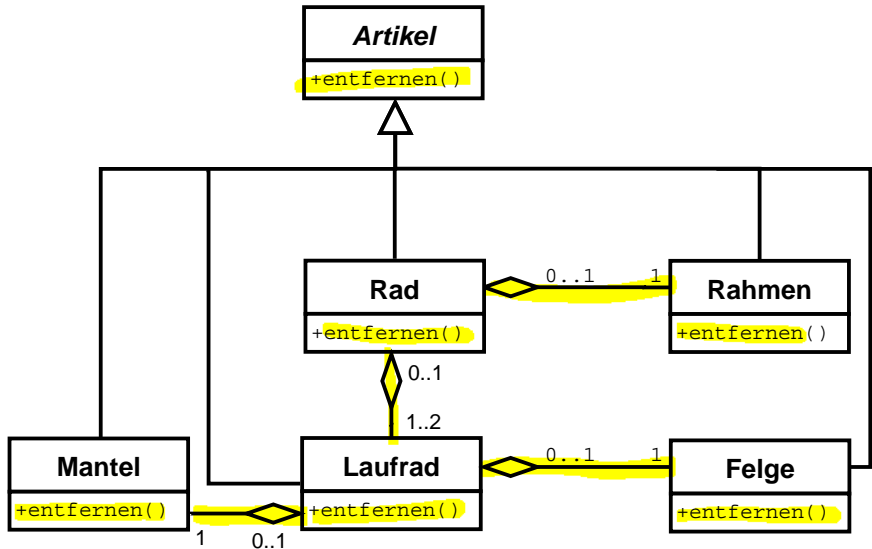


# Beispielentwurfsproblem



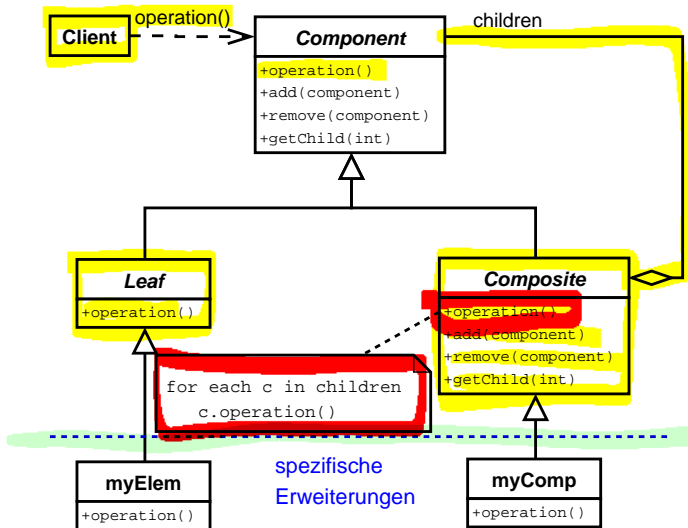
# Beschreibung von Mustern (Gamma u. a. 2003) I

- **Name:** *Composite*
- **Zweck:** Teil-von-Hierarchie mit einheitlicher Schnittstelle beschreiben (überall wo ein Ganzes benutzt werden kann, kann auch ein Teil benutzt werden und umgekehrt)
- **Motivation:** ... Einführung anhand eines konkreten Beispiels...
- **Anwendbarkeit:**
  - wenn Teil-von-Beziehung beschrieben werden soll
  - uniforme Schnittstelle für alle Elemente der Hierarchie

0 Punkte

# Beschreibung von Mustern (Gamma u. a. 2003) II

- Struktur:



- **Teilnehmer:**

- *Component:*

- deklariert einheitliche Schnittstelle
    - implementiert Standardverhalten
    - deklariert Schnittstelle, um seine Teile zu verwalten
    - (optional) definiert Schnittstelle, um Behälter zu erhalten

- *Leaf:*

- repräsentiert atomare Komponente
    - definiert Verhalten für atomare Komponenten

- *Composite:*

- definiert Verhalten für zusammengesetzte Komponenten
    - speichert Teile
    - implementiert Operationen zur Verwaltung von Teilen

- *Client:*

- manipuliert Objekte der Komponenten nur durch die Schnittstelle von *Composite*

- **Kollaborationen:**

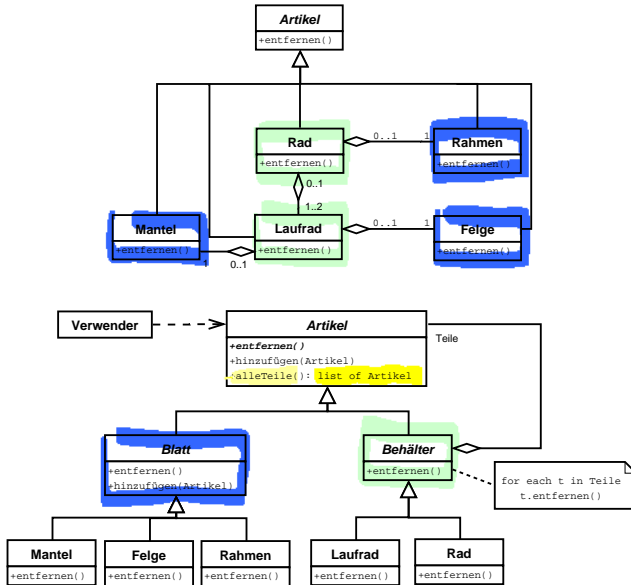
- *Clients* benutzen *Component*-Schnittstelle
- falls Empfänger ein *Leaf* ist, antwortet es direkt
- falls Empfänger ein *Composite* ist, wird die Anfrage an Teile weitergeleitet (möglicherweise mit weiteren eigenen Operationen vor und/oder nach der Weiterleitung)

- **Konsequenzen:**

- zweiteilt die Klassenhierarchie in *Leaf* und *Composite* mit einheitlicher Schnittstelle
- uniforme Verwendung auf Seiten des *Clients*
- neue Komponenten können leicht hinzugefügt werden
- könnte die Struktur unnötig allgemein machen (dynamische versus statische Typisierung)

- **Implementierung:** ... Hinweise zur Implementierung ...

# Dynamische versus statische Typisierung



# Kategorien von Entwurfsmustern

- Erzeugungsmuster
  - betreffen die Erzeugung von Objekten
  - Beispiel: Factory Method
- Strukturelle Muster:
  - betreffen Komposition von Klassen und Objekten
  - Beispiel: Composite
- Verhaltensmuster:
  - betreffen Interaktion und Verantwortlichkeiten
  - Beispiel: Observer



# Erweiterung für andere Domänen

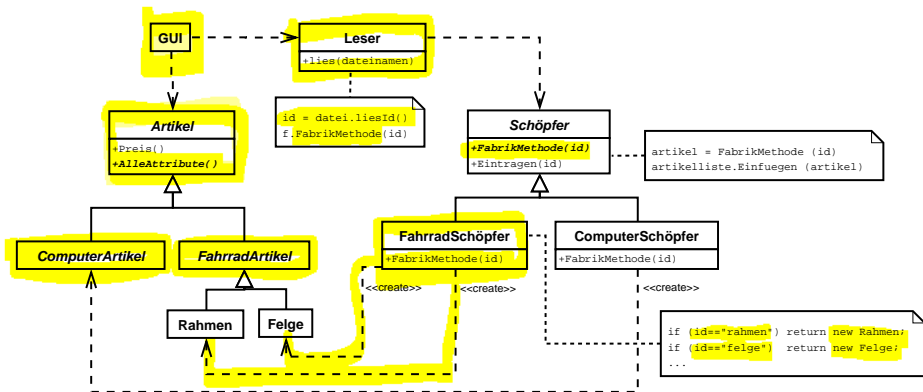
## Anforderungen:

- Artikeldaten sollen von einer Datei gelesen werden können
- zukünftig sollen andere Domänen unterstützen (Fahrrad, Computer und Klettern)
- die Objekte dieser Domänen sind unterschiedlich
- notwendige Anpassungen sollen einfach realisiert werden können

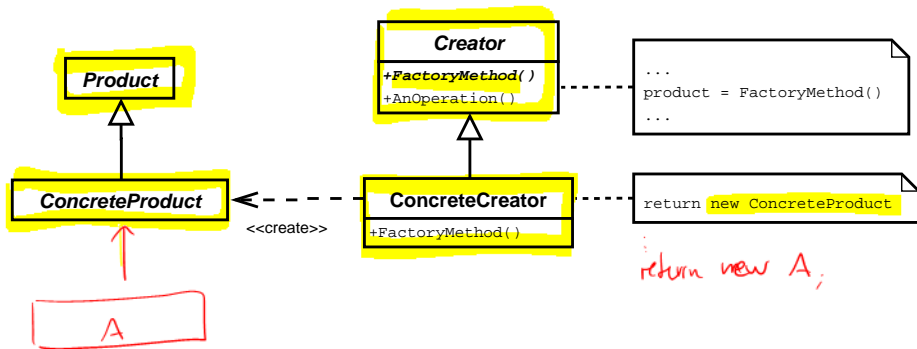
## Lösungsstrategien:

- die Klassen der Benutzungsschnittstelle beziehen sich nur auf die Schnittstelle der abstrakten Klasse Artikel
- Datei hat gleiche Syntax für alle Domänen (nur die Inhalte variieren)
- die Artikel werden beim Einlesen der Datei als Objekte erzeugt
  - aber der Leser muss doch die Konstruktoren der Objekte kennen, oder was?

# Lösungsstrategie



# Entwurfsmuster: *Factory Method*



- eine Klasse weiß in manchen Fällen nicht im Voraus, von welcher Klasse ein zu erzeugendes Objekt sein soll
- die konkreten Unterklassen einer Klasse sollen dies entscheiden
- Verantwortlichkeit wird an Unterklassen delegiert, und das Wissen über die Unterklasse, an die delegiert wird, soll nur an einem Punkt vorhanden sein

- Product
  - deklariert die Schnittstelle von Objekten, die die Fabrikmethode erschafft
- ConcreteProduct
  - implementiert die Product-Schnittstelle
- Creator
  - deklariert die Fabrikmethode
  - (optional) implementiert eine Standardfabrikmethode, die ein spezifisches konkretes Objekt erzeugt
  - kann Fabrikmethode aufrufen, um ein Product-Objekt zu erzeugen
- ConcreteCreator
  - überschreibt die Fabrikmethode, um konkretes Product-Objekt zu erschaffen

# Entwurfsmuster Observer



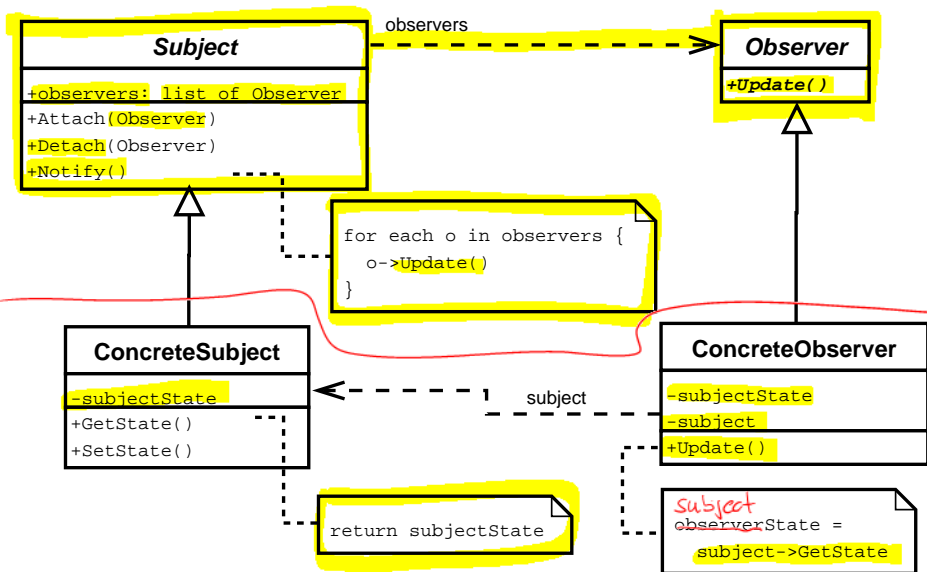
## Anwendbarkeit

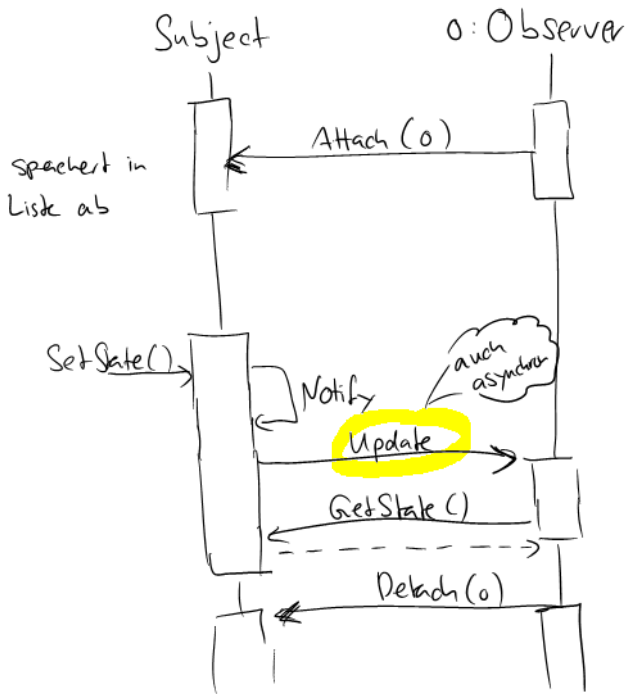
- Komponenten hängen von anderen Komponenten ab
- Änderung der einen Komponente muss Änderung der anderen nach sich ziehen
- Komponenten sollen lose gekoppelt sein: Komponente kennt seine Abhängigen nicht im Voraus (zur Übersetzungszeit)

## Lösungsstrategie

- Abhängige registrieren sich bei Komponente
- Komponente informiert alle registrierten Abhängigen über Zustandsänderung

# Entwurfsmuster Observer: Struktur







# Entwurfsmuster Observer: Teilnehmer

- Subject
  - kennt seine Observer (zur Laufzeit)
  - kann beliebig viele Observer haben
  - stellt Schnittstelle zur Verfügung, um Observer zu registrieren und abzutrennen
- Observer
  - deklariert Schnittstelle für die Update-Nachricht
- ConcreteSubject
  - hat einen Zustand, der ConcreteObserver interessiert
  - sendet Bekanntmachung via Notify(), wenn sich Zustand ändert (SetState)
- ConcreteObserver
  - kennt ConcreteSubject-Objekt
  - verarbeitet Zustand dieses Subjects
  - implementiert Update, um auf veränderten Zustand zu reagieren

# Entwurfsmuster Observer: Konsequenzen

- † • abstrakte Kopplung zwischen Subject und Observer
- † • unterstützt Rundfunk (Broadcast)
- • unerwartete Updates, komplizierter Kontrollfluss
- • viel Nachrichtenverkehr, auch dann wenn sich ein irrelevanter Aspekt geändert hat

# Entwurfsmuster Observer: Verfeinerungen

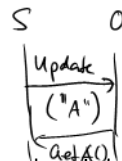
- Push-Modell

- Subject sendet detaillierte Beschreibung der Änderung
  - umfangreiches Update
  - vermeidet GetState(), aber nicht Update()



- Pull-Modell

- Subject sendet minimale Beschreibung der Änderung
  - Observer fragt gegebenenfalls die Details nach
  - erfordert weitere Nachrichten, um Details abzufragen



- Explizite Interessen

- Observers melden Interesse an spezifischem Aspekt an; Aspekt wird zusätzlicher Parameter von Update

