

# Design Patterns

(Software-Architektur)

Prof. Dr. Oliver Braun

Letzte Änderung: 11.07.2017 15:12

- ▶ Gang of Four:

Erich Gamma, Richard Helm, Ralph Johnson & John Vlissides:  
**Design Patterns: Entwurfsmuster als Elemente  
wiederverwendbarer objektorientierter Software.**

# Design Patterns nach GoF

- ▶ Erzeugungsmuster
  - ▶ Abstract Factory
  - ▶ Builder
  - ▶ Factory Method
  - ▶ Prototype
  - ▶ Singleton
- ▶ Strukturmuster
  - ▶ Adapter
  - ▶ Bridge
  - ▶ Composite
  - ▶ Decorator
  - ▶ Facade
  - ▶ Flyweight
  - ▶ Proxy
- ▶ Verhaltensmuster
  - ▶ Chain of Responsibility
  - ▶ Command
  - ▶ Interpreter
  - ▶ Iterator
  - ▶ Mediator

# Aus Lesetexten bekannt

- ▶ Factory Method
- ▶ Decorator

# Abstract Factory

Bereitstellung einer Schnittstelle zum Erzeugen verwandter oder voneinander abhängiger Objektfamilien ohne die Benennung ihrer konkreten Klassen.

Auch bekannt als **Kit**.

# Motivation

- ▶ eine Anwendung auf verschiedenen Plattformen
- ▶ mit plattformspezifischem Look-and-Feel
- ▶ aber keiner Hartcodierung von plattformspezifischen Widgets

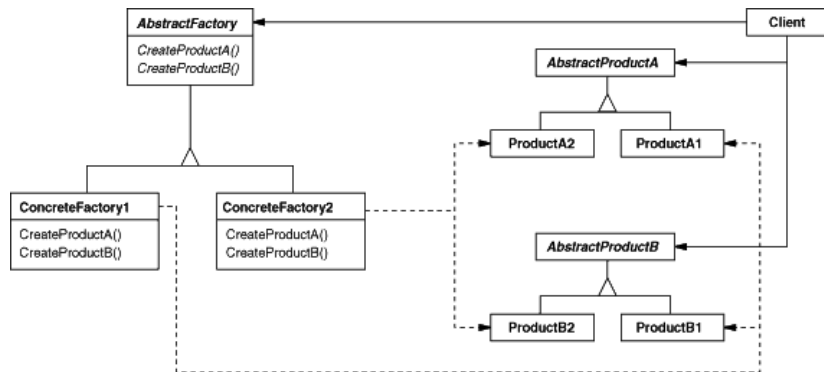
Verwendung empfiehlt sich, wenn

- ▶ System unabhängig von Art der Erzeugung, Komposition und Darstellung seiner Produkte arbeiten soll
- ▶ System mit einer von mehreren Produktfamilien konfiguriert werden soll
- ▶ Familie verwandter Produktobjekte für die gemeinsame Verwendung vorgesehen ist.
- ▶ Klassenbibliothek von Produkten bereitgestellt werden soll, aber nur Schnittstelle nicht Implementierung preis gegeben werden soll.

Produkte meint die Objekte.



# Struktur



Quelle: GoF-Buch

**AbstractFactory** deklariert Schnittstelle für Operationen, die abstrakte Produktobjekte erzeugen.

**ConcreteFactory** implementiert die Operationen zur Erzeugung konkreter Produktobjekte.

- ▶ normalerweise eine einzelne Instanz einer ConcreteFactory-Klasse zur Laufzeit
- ▶ diese erzeugt Objekte mit spezifischer Implementierung
- ▶ AbstractFactory delegiert an ConcreteFactory

# Konsequenzen

1. Isolierung konkreter Klassen.
2. Einfacher Austausch von Produktfamilien.
3. Produktkonsistenz.
4. Unterstützung neuer Produktarten.

## Beispiel

```
/**
 * A Scala Factory Pattern example by Alvin Alexander,
 * slightly adapted by Oliver Braun
 * http://devdaily.com
 */

trait Animal {
  def speak(): Unit
}

object Animal {
  private class Dog extends Animal {
    override def speak(): Unit = println("woof")
  }
  private class Cat extends Animal {
    override def speak(): Unit = println("meow")
  }
  def apply(s: String): Animal =
```

# Adapter

Anpassung der Schnittstelle einer Klasse an ein anderes von den Clients erwartetes Interface. Das Design Pattern *Adapter* ermöglicht die Zusammenarbeit von Klassen, die ansonsten auf Grund der Inkompatibilität ihrer Schnittstellen nicht dazu in der Lage wären.

Auch bekannt als **Wrapper**.

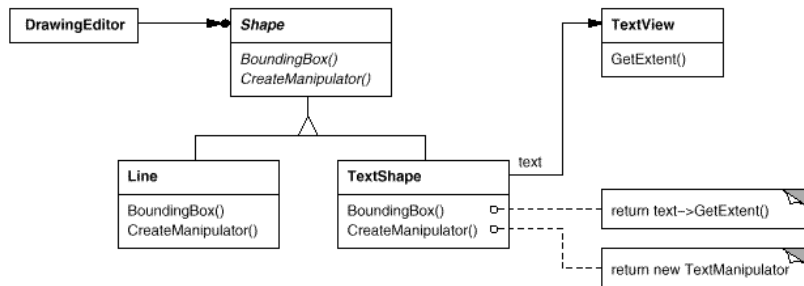
Mitunter ist die Nutzung einer grundsätzlich als wiederverwendbar entwickelten Toolkit-Klasse nur deshalb nicht möglich, weil sie nicht der domainspezifischen Schnittstelle entspricht, die eine Anwendung voraussetzt.

Verwendung empfiehlt sich, wenn

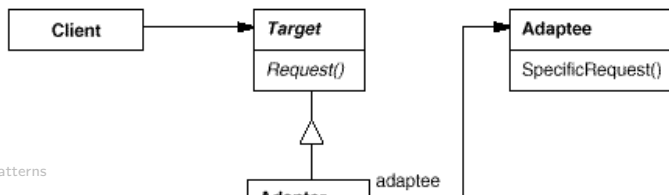
- ▶ existierende Klasse genutzt werden soll, deren Schnittstelle den aktuellen Anforderungen nicht genügt.
- ▶ eine wiederverwendbare Klasse erzeugt werden soll, die mit voneinander unabhängigen und nicht vorhersehbaren Klassen zusammenarbeitet, also Klassen deren Schnittstellen gegebenenfalls nicht kompatibel sind.
- ▶ mehrere bereits vorhandene Unterklassen verwendet werden sollen, es aber unpraktisch wäre, deren individuellen Schnittstellen durch Unterklassenbildung anzupassen. Hier schafft ein *Objektadapter* Abhilfe, in dem er die Schnittstelle seiner Basisklasse adaptiert.



# Struktur



Mehrfachvererbung beim Klassenadapter, Quelle: GoF-Buch



Die Clients rufen die Operationen auf einer Adapter-Instanz auf, und der Adapter ruft die Adaptee-Operationen auf, die den Request ausführen.

# Konsequenzen

## ▶ Klassenadapter

- ▶ passt das zu adaptierende Objekt durch Zuweisung einer konkreten Adaptee-Klasse an die Target-Schnittstelle an.
- ▶ funktioniert nicht, wenn Klasse mitsamt all Ihren Unterklassen adaptiert werden soll.
- ▶ gestattet dem Adapter, das Verhalten des adaptierten Objekts zu überschreiben
- ▶ benutzt lediglich ein einziges Objekt

## ▶ Objektadapter

- ▶ ermöglicht einem einzelnen Adapter die Zusammenarbeit mit mehreren adaptierten Objekten
- ▶ kann auch alle adaptierten Objekte um neue Funktionalität erweitern
- ▶ erschwert das Überschreiben des Verhaltens des Adaptee-Objektes

## Beispiel in Java

Quelle: <https://pavelfatin.com/design-patterns-in-scala/>

```
public interface Log {
    void warning(String message);
    void error(String message);
}

public final class Logger {
    void log(Level level, String message) { /* ... */ }
}

public class LoggerToLogAdapter implements Log {
    private final Logger logger;

    public LoggerToLogAdapter(Logger logger) {
        this.logger = logger;
    }
}
```

## Beispiel in Scala

Quelle: <https://pavelfatin.com/design-patterns-in-scala/>

*In Scala, we have a built-in concept of interface adapters, expressed as implicit classes:*

```
trait Log {  
  def warning(message: String)  
  def error(message: String)  
}  
  
final class Logger {  
  def log(level: Level, message: String) { /* ... */ }  
}  
  
implicit class LoggerToLogAdapter(logger: Logger) extends Log {  
  def warning(message: String): Unit =  
    logger.log(WARNING, message)  
  def error(message: String): Unit =  
    logger.log(ERROR, message)  
}
```

## Stackable Trait

# Stackable Trait Pattern in Scala

- ▶ ähnlich dem Decoration Pattern
- ▶ es werden aber keine Objekte dekoriert, sondern Klassen/Traits zur Compilierzeit
- ▶ Beispiel: siehe Livecoding-Repo

## Cake Pattern



# Dependency Injection

- ▶ Konzept der OOP
- ▶ Abhängigkeiten eines Objektes werden zur Laufzeit geregelt
- ▶ benötigt ein Objekt bei seiner Initialisierung ein anderes
  - ▶ wird dies nicht vom Objekt selbst erzeugt
  - ▶ sondern von außen injiziert, d.h. es ist z.B. an einer zentralen Stelle hinterlegt
- ▶ in Scala kann dies elegant mit sog. Self Annotations und dem Cake Pattern umgesetzt werden
- ▶ Beispiel: siehe Livecoding-Repo

# Vorteile Self-Annotations gegenüber Vererbung

## ▶ Vererbung

```
trait A
trait B extends A
trait C extends B
```

- ▶ aus C kann auf alle Member von A (geerbt) direkt zugegriffen werden

## ▶ Self-Annotations

```
trait A
trait B {this: A =>}
trait C {this: B => }
```

- ▶ aus C kann nur auf Member von B zugegriffen werden (bessere Kapselung)

## ▶ GoF:

- ▶ Favor 'object composition' over 'class inheritance'.