Software Engineering II (IB)
Design Patterns

Prof. Dr. Oliver Braun

Letzte Änderung: 16.05.2017 20:56

Standardwerk

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

Erzeugungsmuster

- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton

Strukturmuster

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

Verhaltensmuster

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

Abstract Factory

Zweck

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

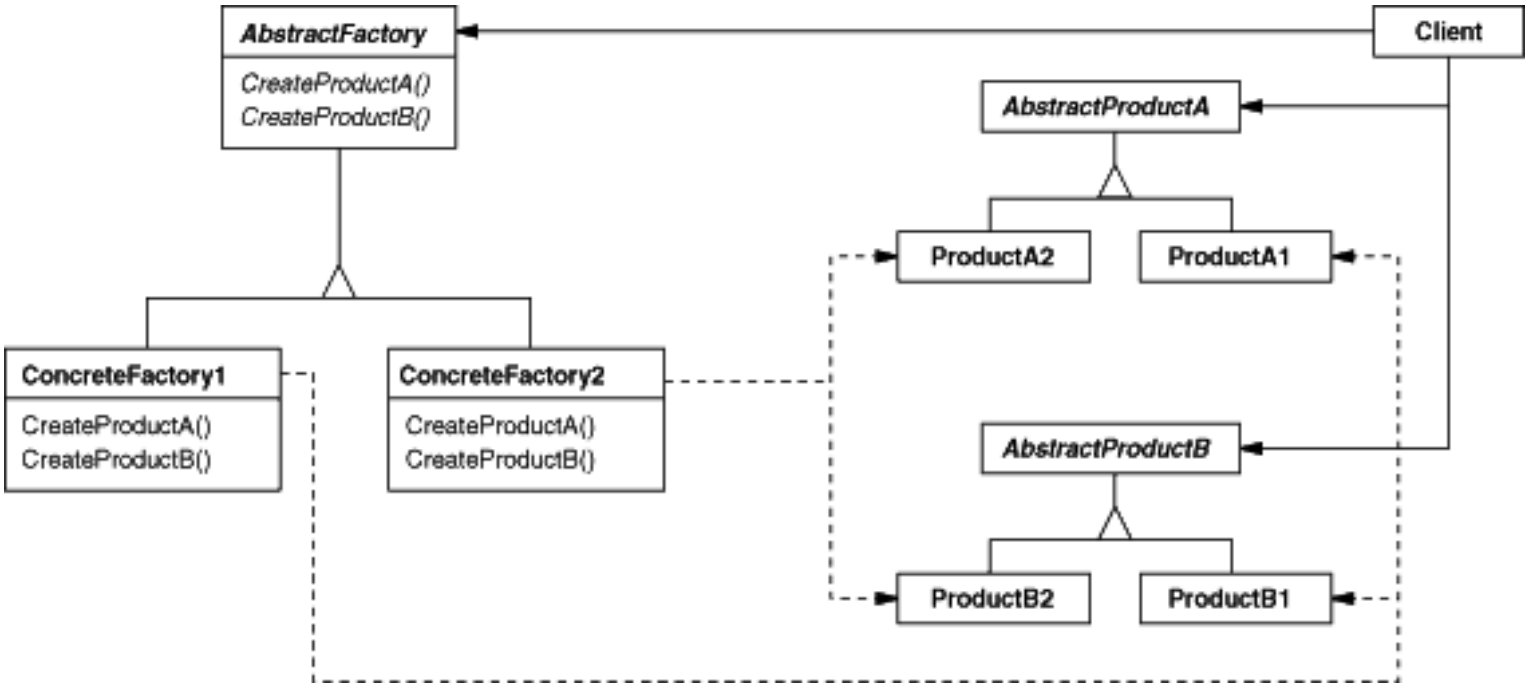
Anwendbarkeit

Verwendung empfiehlt sich, wenn

- System unabhängig von Art der Erzeugung, Komposition und Darstellung seiner Produkte arbeiten soll
- System mit einer von mehreren Produkfamilien 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

Bedeutung der Klassen

AbstractFactory deklariert Schnittstelle für Operationen, die abstrakte Produktobjekte erzeugen.

ConcreteFactory Implementiert die Operationen zur Erzeugung konkreter Produktobjekte.

AbstractProduct Deklariert eine Schnittstelle für einen bestimmten Produktobjekttyp.

ConcreteProduct Definiert ein von der zugehörigen konkreten Factory zu erzeugendes Produktobjekt.

Client Verwendet ausschließlich von `AbstractFactory` und `AbstractProduct` deklarierte Schnittstellen.

Interaktionen

- 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

```
trait Product {
  val name: String
  def price: Int
  override def toString(): String =
    name + ", €" + price/100 + "," + price%100
}
```

Beispiel — Fortsetzung

```
object Product {
  private class Pizza extends Product {
    override val name = "Pizza"
    override def price = 795
  }
  private class Coke extends Product {
    override val name = "Coke"
    override def price = 285
  }
  def apply(s: String): Product = s match {
    case "Pizza" => new Pizza
    case "Coke"  => new Coke
  }
}
```

Beispiel — Fortsetzung

```
object Shop extends App {
  val pizza = Product("Pizza")
  val coke = Product("Coke")
  ...
}
```

Adapter

Zweck

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**.

Motivation

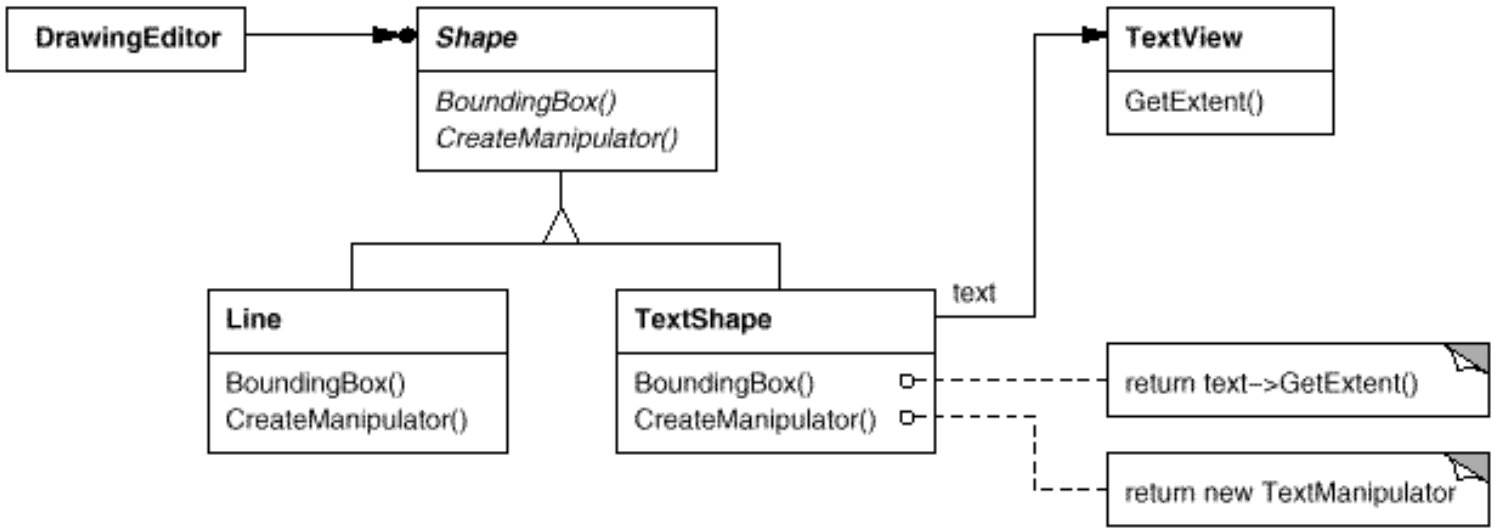
Mitunter ist die Nutzung einer grundsätzlich als wiederverwendbar entwickelten Toolkit-Klasse nur deshalb nicht möglich, weil sie nicht der domainspezifischen Schnittstelle entspricht, die eine Anwendung voraussetzt.

Anwendbarkeit

Verwendung empfiehlt sich, wenn

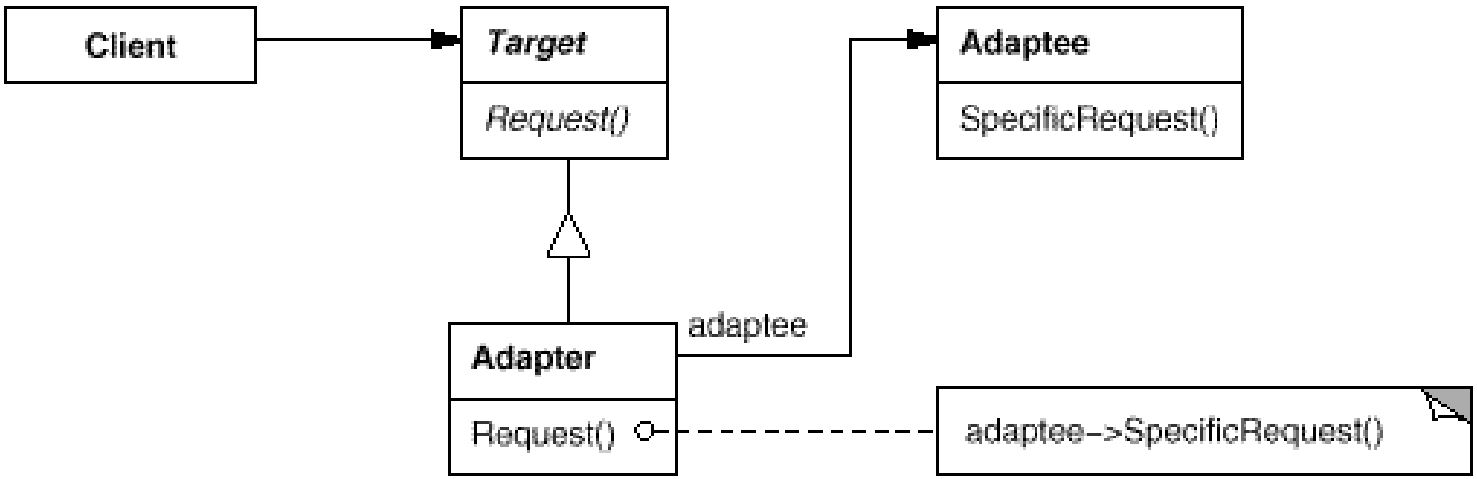
- existierende Klasse genutzt werden soll, deren Schnittstelle den aktuellen Anforderungen nicht genügt.
- eine wiederverwendbare Klasse erzeugt werden soll, die mit voneinander unabhängigen und nicht vorhersehbaren Klassen zusammenarbeitet, also Klassen deren Schnittstellen gegebenenfalls nicht kompatibel 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 Klassenadapter



Mehrfachvererbung beim Klassenadapter, Quelle: GoF-Buch

Struktur Objektadapter



Objektkomposition beim Objektadapter, Quelle: GoF-Buch

Bedeutung der Klassen

Target	Definiert die vom Client verwendete domainspezifische Schnittstelle.
Client	Arbeitet mit Objekten zusammen, die der `Target`-Schnittstelle entsprechen.
Adaptee	Definiert eine existierende Schnittstelle, die adaptiert werden muss.
Adapter	Adaptiert die `Adaptee`-Schnittstellen an die `Target`-Schnittstelle.

Interaktionen

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) { /* ... */ }
}
```

Beispiel in Java — Fortsetzung

```
public class LoggerToLogAdapter implements Log {
    private final Logger logger;

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

    public void warning(String message) {
        logger.log(WARNING, message);
    }

    public void error(String message) {
        logger.log(ERROR, message);
    }
}
Log log = new LoggerToLogAdapter(new 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) { /* ... */ }
}
```

Beispiel in Scala — Fortsetzung

```
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)
}

val log: Log = new Logger()
```

Stackable Trait

Stackable Trait Pattern in Scala

- ähnlich dem Decoration Pattern
- es werden aber keine Objekte dekoriert, sondern Klassen/Traits zur Compilierzeit

Beispiel: Stackable Pizza

```
trait PizzaT {
  def price: Int
  def description: String
  def priceToString: String = {
    val cent = price%100
    "€" + price/100 + "," +
      (if (cent < 10) "0"+cent else cent)
  }
}
class Pizza extends PizzaT {
  override def price = 500
  override def description =
    "Pizza with tomato and cheese"
}
```

Beispiel: Mit extra Käse

```
trait ExtraMozzarellaCheese extends Pizza {
  override def price = super.price + 50
  override def description =
    super.description + "\n + extra mozzarella cheese"
}
```

Beispiel: Pizza mit extra Käse

```
object PizzaShop extends App {
  val pizza = new Pizza with ExtraMozzarellaCheese
  println(pizza.description)
  println(pizza.priceToString)
}
```

Ausgabe:

```
Pizza with tomato and cheese
 + extra mozzarella cheese
€5,50
```

Beispiel: Und noch mehr Extras

```
trait Mushrooms extends Pizza {
  override def price = super.price + 70
  override def description =
    super.description + "\n + mushrooms"
}
trait Chillies extends Pizza {
  override def price = super.price + 105
  override def description =
    super.description + "\n + chillies"
}
trait Seafood extends Pizza {
  override def price = super.price + 250
  override def description =
    super.description + "\n + seafood"
}
```

Beispiel: Mmmh lecker

```
object PizzaShop extends App {
  val pizza1 = new Pizza with Mushrooms
                         with Seafood
  println(pizza1.description)
  println(pizza1.priceToString)
  val pizza2 = new Pizza with ExtraMozzarellaCheese
                         with Chillies
                         with Seafood
  println(pizza2.description)
  println(pizza2.priceToString)
}
```

Beispiel: Her damit...

Ausgabe:

```
Pizza with Tomato and Cheese
 + mushrooms
 + seafood
€8,20
Pizza with Tomato and Cheese
 + extra mozzarella cheese
 + chillies
 + seafood
€9,05
```

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

- wir brauchen mehrere `UserDAO`-Implementierungen, die wir im `UserService` nutzen können
  - H2
  - PostgreSQL für Heroku
  - ...
- wir wollen uns erst in der `App` für eine davon entscheiden und nur eine `UserService`-Implementierung schreiben
- die Implementierung soll nicht über einen Konstruktor übergeben, sondern als Abhängigkeit injiziert werden (engl. Dependency Injection)

Beispiel: UserDAO-Traits

Die beiden Implementierungen mit einer Generalisierung

```
trait GenericUserDao {
  def imp: String
  ...
}

trait H2UserDao extends GenericUserDao {
  override def imp: String = "H2 Implementation"
  ...
}

trait HerokuPostgreSUserDao extends GenericUserDao {
  override def imp: String =
    "HerokuPostgreSDao Implementation"
  ...
}
```

Beispiel

```
trait UserServiceTrait {
  dao: GenericUserDao => // self type annotation
  ...
}
```

Beispiel

```
object UserServices extends App {
  val svc = new UserServiceTrait with H2UserDao
  println("svc using H2 DI returns:\n    " + svc.imp)
  val svc2 = new UserServiceTrait
                  with HerokuPostgreSUserDao
  println("svc2 using Heroku PostgreSQL DI returns:"
          + "\n    " + svc2.imp)
}
```

Ausgabe:

```
svc using H2 DI returns:
    H2 Implementation
svc2 using Heroku PostgreSQL DI returns:
    HerokuPostgreSDao Implementation
```