Spring-Framework einfach lernen

Professionelle Software-Entwicklung

Dr.-Ing. Stefan Koch

15.08.2011 Version 0.1

Inhalt

1 VORWORT – NOCH EIN BUCH ZU SPRING? ... 3

2 STAND DER TECHNIK ... 4

2.1 ARCHITEKTUR EINER UNTERNEHMENS-ANWENDUNG ... 4

2.2 ARCHITEKTUR DER BUSINESS LAYER ... 5

2.3 SERVICE LAYER: SERVICE PATTERN ... 5

2.4 SERVICES ZUR VERFÜGUNG STELLEN: FACTORY PATTERN ... 6

2.5 DAS DAO-PATTERN ... 7

3 DIE PRODUKTION VON SPRING-BEANS ... 9

3.1 ERSTES BEISPIEL ... 9

3.2 DAS LEBEN EINER SPRING-BEAN ... 10

4 DAS KNÜPFEN VON OBJEKT-NETZEN ... 12

4.1 ATTRIBUTE ÜBER SETTER INJIZIEREN ... 12

4.1.1 Abgekürzte Schreibweise ... 12

4.2 AUSWERTUNG VON PROPERTY-DATEIEN ... 13

4.3 SPRING-BEANS MIT PARAMETRISIERTEN KONSTRUKTOREN ERZEUGEN ... 13

4.3.1 Passenden Konstruktor auswählen ... 14

4.4 INJEKTION VON OBJEKTEN ... 14

4.5 COLLECTIONS ALS PARAMETER ÜBERGEBEN ... 15

5 SPRING-FACTORY IM EINSATZ ... 17

5.1 DIE SERVICELOCATORFACTORY IM ÜBERBLICK ... 20

6 ASPEKTORIENTIERTER ANSATZ ... 21

6.1 FACHBEGRIFFE DER ASPEKTORIENTIERTEN PROGRAMMIERUNG ... 21

6.1.1 Aspekt ... 21

6.1.2 Joinpoint ... 21

6.1.3 Advice ... 21

6.2 PRINZ DER ASPEKTORIENTIERTEN PROGRAMMIERUNG MIT DEM SPRING-FRAMEWORK ... 21

6.3 KONFIGURATION DER ASPEKTORIENTIERTEN PROGRAMMIERUNG ... 24

6.3.1 Advice-Interfaces ... 24

6.3.2 Interface MethodInterceptor ... 24

6.3.3 Advisor-Konfiguration ... 25

REFERENCES / LITERATURVERZEICHNIS ... 26

1 Vorwort – noch ein Buch zu Spring?

21:55 Uhr – wieso ein Buch zum Spring Framework, Version 3, schreiben? Zum Spring Framework [1] gibt es ein gelungenes Reference-Manual [1] – genauso kostenfrei, wie das Spring Framework selbst. Die Google-Suche [3] ergibt beim Suchbegriff *Spring Framework book* ungefähr 3,6 Millionen Ergebnisse.

Morgen wird das Spring Framework bestimmt durch eine weiterentwickelte JEE-Spezifikation ersetzt – morgen.

Gut – ich möchte nur ein ganz kleines Buch schreiben. Eins mit ein paar Kochrezepten zu den Standardsituationen der Software-Entwicklung.

Wieso? Mich treibt die Überzeugung. Die Überzeugung, dass der Einsatz des Spring Frameworks zu einer klaren Software-Architektur führen kann. Also zu dem Ziel, bei der der Software-Architekt nachts ruhig schlafen kann, weil er weiß, wie die Entwicklung morgen weiter geht. Einem Ziel bei dem der Entwickler sich auf die Geschäftslogik konzentriert und sich wenig Gedanken über Architektur machen muss.

Für die Implementierung einer typischen Enterprise-Anwendung, die meist als Java-Enterprise-Anwendung umzusetzen ist, sind die Ideen schnell beschrieben und leicht umgesetzt – daher nur ein ganz kleines Buch.

Die Ideen sind nicht auf meinem Mist gewachsen! Sie sind in der Entwicklerwelt weit verbreitet – und doch in erstaunlich vielen Projekten nicht anzutreffen.

Mir wurden sie durch das Spring Framework nahegelegt, weswegen Ehre und Dank den Erfindern des Spring Frameworks zustehen.

Besonders dankbar bin ich auch den Kollegen im In- und Ausland, die in gemeinsamen Projekten erheblich zu meinem Erfahrungsschatz beigetragen haben.

Das Manuskript wird nach und nach erweitert. Anregungen und Wünsche können Sie richten an kontakt@einfach-lernen.info.

2 Stand der Technik

Aufgabe der Softwareentwicklung ist es, Geschäftslogik umzusetzen. In sehr vielen Anwendungen werden die dazu erforderlichen Daten in einer relationalen Datenbank abgelegt.

Die Umgebung, in der die Geschäftslogik umgesetzt wird, ist meist vorgegeben: Nach dem JEE-Standard kommen ein EJB- oder Web-Container zum Einsatz.

Unser Ziel ist es, eine gut test- und wartbare Software zu schaffen. Dazu leistet das Spring Framework einen wichtigen Beitrag.

Um die Ansätze der Software-Architektur zu verstehen, sollten Sie die folgenden Abschnitte des Kapitels durchlesen.

2.1 Architektur einer Unternehmens-Anwendung

Betrachtet werden übliche Anforderungen einer Enterprise-Anwendung. Die Enterprise-Anwendung ist gekennzeichnet durch den Zugriff auf ein Enterprise Information System (EIS), in der Regel also auf eine Unternehmensdatenbank.

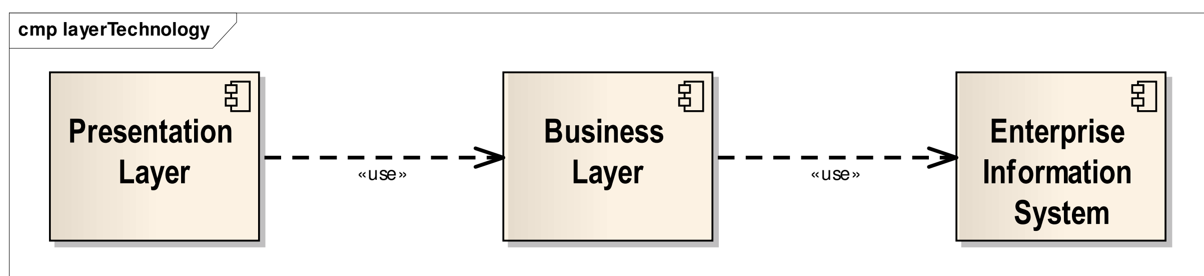


Abbildung 1: Geschäftsanwendungen werden üblicherweise in Schichten aufgeteilt.

Abbildung 1 beschreibt die Komponenten, die eine Geschäftsanwendung mit Benutzerschnittstelle haben sollte.

Die **Presentation Layer** sorgt für die Darstellung und das Verhalten der Benutzerschnittstelle. Die **Business Layer** realisiert die Geschäftslogik. Sie ist die zentrale Komponente einer Geschäftsanwendung. Sie übernimmt die Verantwortung dafür, dass die Datenbank (das Enterprise Information System) einen konsistenten Stand hat.

Das **Enterprise Information System** wird in der Regel durch ein Relationales Management System realisiert.

Die Aufteilung der Anwendung in diese Schichten bringt eine Verringerung der Abhängigkeiten mit sich. Jede Komponente ist nur von seiner rechten Komponente abhängig.

Die Verringerung der Abhängigkeiten führt dazu, dass

- Tests einfacher durchgeführt werden können. Ein Dialog kann dadurch getestet werden, dass anstelle der Implementierung der Business Layer die erforderlichen Methoden als Mocks realisiert werden, die nur so tun, als würden sie Geschäftslogik umsetzen.
- Fehler leichter lokalisiebar sind.
- Veränderungen des Codes (Fehlerbehebungen, Anpassungen) geringere Auswirkungen haben. Damit sinkt die Wahrscheinlichkeit von Folgefehlern.
- Aufgaben sich verteilen lassen. Die Komponenten lassen sich beinahe unabhängig voneinander entwickeln. Das Verhalten der Komponenten wird dafür durch Schnittstellen festgelegt. Dieses ist besonders bei einem größeren Projekt für die Aufgabenverteilung von großer Bedeutung.
- Teile wiederverwendbar sind: Geschäftslogik, die von der Benutzerschnittstelle verwendet wird, kann genauso gut durch eine Business-2-Business-Schnittstelle ausgeführt werden.

Der Nutzen der Schichten-Architektur ist unbestritten. Die Struktur innerhalb dieser Schichten ist Gegenstand weiterer Architektur-Entscheidungen.

2.2 Architektur der Business Layer

Ein Architektur-Muster für die Business-Layer besteht aus den drei Komponenten.

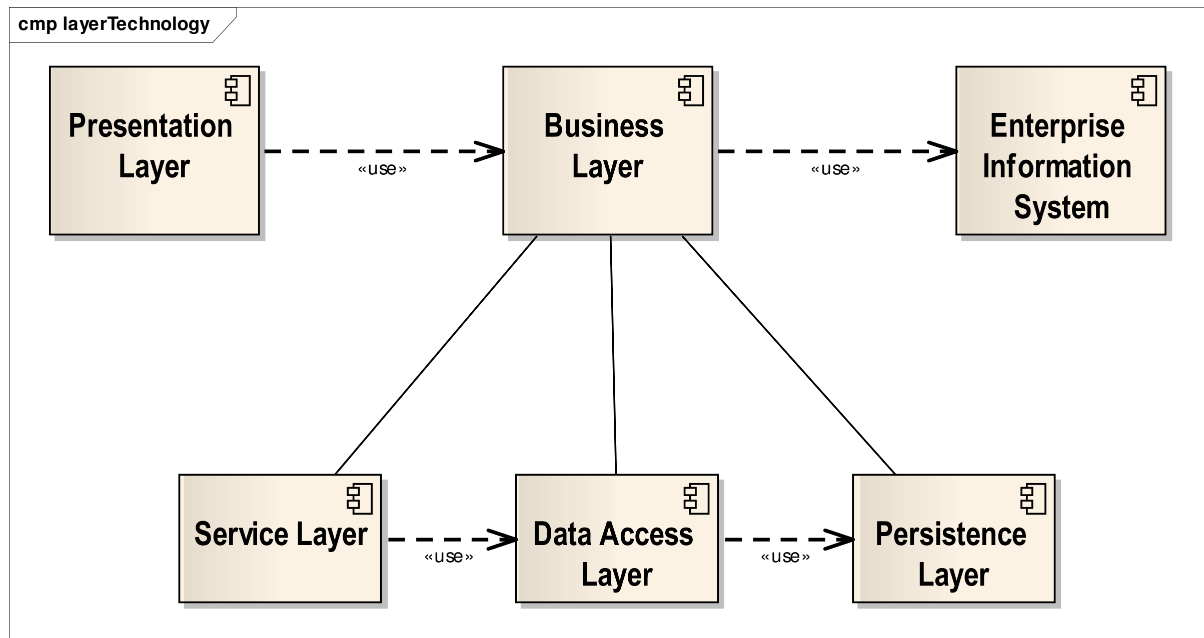


Abbildung 2: Strukturierung der Business Layer

- **Service Layer**: In dieser Schicht werden Anwendungsfälle, wie Kunde anlegen oder Bestellung durchführen implementiert.
- **Data-Access Layer**: Diese Schicht ist spezialisiert auf den Datenzugriff. In welchen Tabellen sind welche Daten abgelegt? Welche Beziehungen zwischen Tabellen müssen berücksichtigt werden? Wie können Zugriffe performant erfolgen?
- **Persistence Layer**: In dieser Schicht findet sich oftmals ein Framework für das objekt-relationale Mapping. Objekte werden in Tabellen abgelegt und umgekehrt. Im Regelfall wird dazu ein geeignetes Framework (z B. Hibernate [6]) eingesetzt.

Neben den im Abschnitt 2.1 dargestellten Vorteile, die eine Schichtenarchitektur mit sich bringt, lassen sich in diesem Fall weitere Vorteile aufführen:

- Es gibt unterschiedliche Aspekte bei der Entwicklung:
  - Verständnis der Geschäftslogik: Konzentration auf die Fachlichkeit.
  - Kenntnis der Persistenz: Wo können wie Daten ermittelt oder abgelegt werden?
  - Technische Aspekte beim Zugriff auf das Enterprise Information System.
- Wiederverwertbarkeit: Eine Data-Access-Layer ist für unterschiedliche Geschäftsfälle verwendbar: Es ist leicht vorstellbar, dass das Laden von Kundendaten in sehr vielen Anwendungsfällen benötigt wird.
- Strukturierung, um Transaktionen abzugrenzen: Transaktionen sind oftmals an Anwendungsfälle gebunden: Der Bestellvorgang wird eine Transaktion sein. Wenn irgendwas bei dem Bestellvorgang nicht klappt, so müssen alle Veränderungen wieder rückgängig gemacht werden. Transaktionen sind demnach eine Eigenschaft des Service Layers!

Für die Entwicklung der Schichten haben sich nachfolgende Entwurfsmuster bewährt.

2.3 Service Layer: Service Pattern

Gemogelt – eigentlich gibt es dieses Pattern nicht. Dennoch ist es sinnvoll, zu überlegen, welche Eigenschaften Service-Klassen haben sollen.

Zielsetzung ist die Strukturierung von Methoden, die Anwendungsfälle realisieren. Services sollen modular sein: Ein Service soll einen anderen Service aufrufen können.

Die Strukturierung wird durch die Einführung von Service-Klassen realisiert. Eine Klasse fasst Services eines Themengebietes zusammen.

Die Struktur der Service-Schicht sollte so gewählt werden, dass der Entwickler intuitiv bei der Umsetzung eines Anwendungsfalls weiß, in welcher Klasse die entsprechenden Methoden umzusetzen sind.

Möglicherweise bedient man sich der Persistenz-Objekte. Dann entstehen Klassen wie KundenService, AuftragsService oder ProduktService.

Alternativ bietet sich die Gruppierung nach Anwendungsfälle an, so dass Klassen wie BestellService oder StornierungsService entstehen.

In umfangreichen Projekten wird es möglicherweise Services unterschiedlicher Kategorien geben. Beispielsweise solche, die einen Vorgang steuern und solche, die Basis-Funktionalität ausführen.

Beispiel: Sequenzdiagramm für einen Anwendungsfall

Bei der Implementierung von Service-Klassen wird ein prozeduraler Ansatz verfolgt: Service-Objekte verfügen über keinen Zustand.

2.4 Services zur Verfügung stellen: Factory Pattern

Sie werden sehen, dass in der Service-Schicht eine große Menge an Klassen zusammenkommen. Diese Klassen werden oftmals Beziehungen zueinander haben. Ein BestellService wird wahrscheinlich auf ProduktService, KundenService und LagerService zurückgreifen.

Früher war es ein Problem – jetzt ist es eine Herausforderung:

- Wer sorgt dafür, dass die Objekte in der richtigen Reihenfolge erstellt werden?
- Wo sollen die Objekte überhaupt erstellt werden?
- Wer behält den Überblick über das Objektgeflecht?

All diese Wünsche werden durch Einsatz des Design Patterns Factory[1] erfüllt:

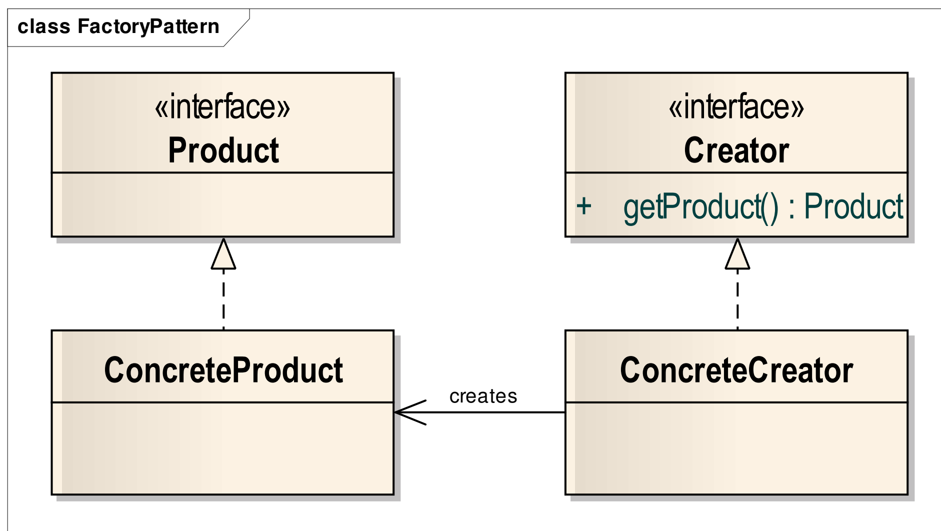


Abbildung 3: Klassendiagramm zum Design Pattern

Beim Factory-Pattern gibt es einen *Creator*, der ein *Product* herstellen kann. Ein Nutzer wird die Methode *getProduct* aufrufen, um ein angefertigtes Produkt zu bekommen.

Der *Creator* fertigt dabei ein *ConcreteProduct* an. Der Nutzer kennt aber nur die Eigenschaft des Produkts, die über das Interface angegeben werden.

Soll das Produkt ein Service sein, so hat der *Creator* vor allem folgende Aufgaben:

- Das Service-Objekt ist unmittelbar einsetzbar. Alle erforderlichen Abhängigkeiten des Objekts müssen dazu vom *Creator* befriedigt werden.
- Die Abhängigkeiten eines Service-Objekts müssen so konfigurierbar sein, dass sie auch bei einer großen Zahl von Service-Objekten noch leicht nachvollziehbar sind.

[1] Wahrscheinlich geht dieses Design Pattern zurück auf die sogenannten Gang of Four [4].

- Als Service-Objekt sollte das Objekt nur einmal erstellt werden.

Der Einsatz des Factory-Patterns gehört – genauso wie die Einteilung der Software in unterschiedlichen Schichten zum Standard.

2.5 Das DAO-Pattern

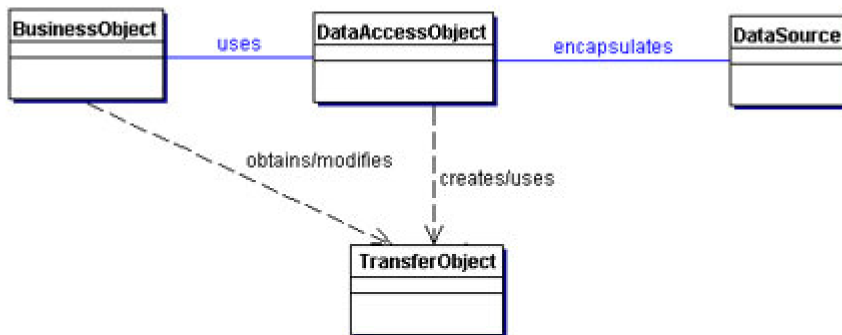
Data Access Objects sind spezielle Service-Klassen. Ihre Aufgabe ist es, den Zugriff zur Persistenz-Schicht zu kapseln.

Diese Service-Spezialisten sind erforderlich, da die Art und Weise, wie auf abgespeicherte Daten zugegriffen wird, viele Aspekte beinhaltet. Beispielsweise:

- In welchem System sind die Daten abgespeichert? Möglicherweise sind mehrere relationale Datenbanken beteiligt oder andere Systeme zur Datenhaltung.
- Mit welchem Treiber/Framework kann auf die Daten zugegriffen werden?
- Wie wird mit einem konkurrierenden Zugriff umgegangen?

Grund genug, diese speziellen Service-Klassen, Data Access Objects, zu schaffen.

Das Prinzip wird beispielsweise in [5] dargestellt.



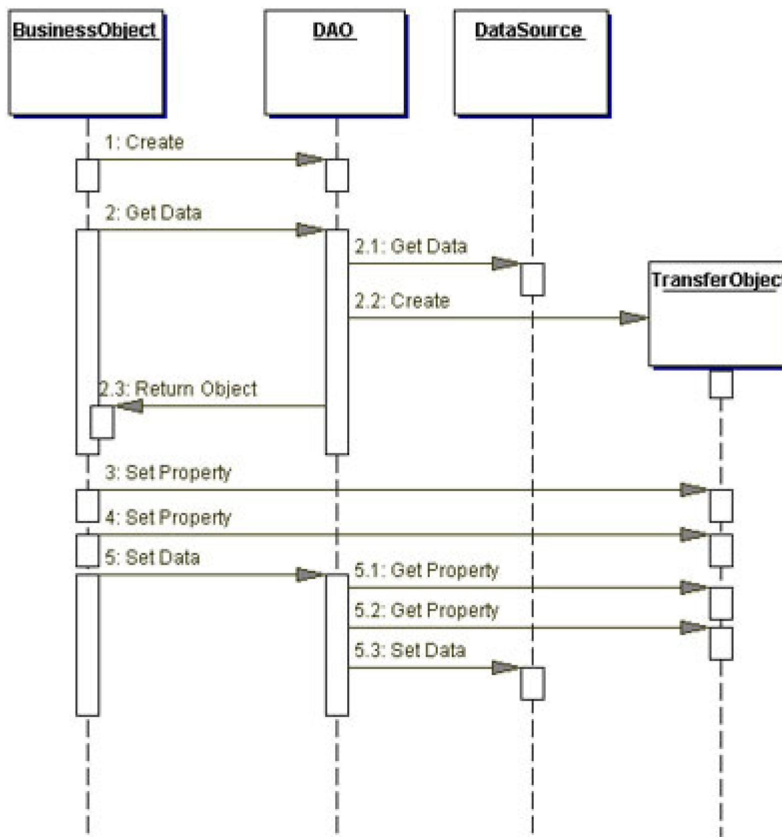


Abbildung 4: Klassen und Sequenzdiagramm zum Entwurfsmuster Data Access Object [5]

In dem Entwurfsmuster gibt es die folgenden Rollen:

- **Businessobject**: Dieses Objekt ist der Konsument. Es möchte Daten von der Datenbank laden oder speichern. In unserer Architektur wird dieses eine Service-Klasse sein.
- **Data Access Object**: Seine Aufgabe ist es, die entsprechenden Lade- oder Speicher-Dienste zur Verfügung zu stellen.
- **Transferobject**: Dieses Objekt beinhaltet die Daten, die geladen oder gespeichert werden sollen. Es werden meist Domain-Objekte wie Kunde, Bestellung verwendet. So kann das *Businessobject* mit den *Transferobject* direkt weiter arbeiten.
- **DataSource** repräsentiert ein Objekt für den Zugriff auf das Enterprise Information System. Bei relationalen Datenbanken kann die *DataSource* eine Connection oder ein Ressource sein, die mit Hilfe einesFrameworks wie Hibernate [6], den Zugriff auf die Daten ermöglicht.

Im Sequenzdiagramm des Architekturmusters erzeugt das *Business-Objekt* ein *DAO* und fordert von diesem über die Methode *getData* Daten an. Das DAO holt diese Daten von der *DataSource*. Die Daten werden in ein *Transferobject* übertragen und dem *Businessobject* übergeben.

Beim Speichern von Daten wird das *Transferobject* durch das *Businessobject* erstellt und dem *DAO* übergeben. Die darin gespeicherten Daten werden der *Datasource* zum Speichern übergeben.

3 Die Produktion von Spring-Beans

Im Kapitel **Fehler! Verweisquelle konnte nicht gefunden werden.** wurden Architektur-Merkmale dargestellt, die zum Stand der Technik gehören. Das Spring-Framework unterstützt bei der Implementierung der Standards.

Dieses Kapitel zeigt, wie mit Hilfe des Spring-Frameworks das Factory-Pattern (s. Abschnitt 2.4) umgesetzt wird.

Am Beispiel einer Bestellung wird diese Vernetzung dargestellt. Da gibt es einerseits die Serviceklasse Bestellvorgang. Diese bietet die Methode bestellungStarten mit den Parametern Kunden- und Produktname an. Beim Aufruf der bestellungStarten-Methode wird die Methode createBestellung des BestellungService-Objekts aufgerufen. Dieses holt mit Hilfe der Klassen BestellungDao und KundenDao die Objekte Produkt und Kunde aus der Datenbank geholt. Es wird ein Bestellung-Objekt erzeugt und zurückgegeben.

Sequenzdiagramm für bestellungStarten.

Der erste Schritt des Bestell-Vorgangs zeigt, dass das Objekt Bestellvorgang Gebrauch macht von dem Objekt BestellService. Dieses wiederum benötigt BestellungDao und KundenDao für die Umsetzung der Geschäftslogik.

3.1 Erstes Beispiel

Zentraler Bestandteil des Springframeworks ist eine Factory. Diese Factory wird durch eine Konfigurationsdatei – meist *applicationContext.xml* genannt – konfiguriert.

Die Factory wird im folgenden Beispiel genutzt, um einen Service für mathematische Aufgaben – den *MathService* - zur Verfügung zu stellen.

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
    <bean id="mathService" class="beispiel01.MathServiceImpl" />
</beans>
```

Listing 1:Erstes Beispiel für eine applicationContext.xml

Die Konfigurationsdatei *applicationContext.xml* ist im XML-Format zu formulieren. Daher ist der XML-Prolog in Zeile 1 erforderlich. Empfehlenswert ist die Verwendung des UTF-8-Zeichensatzes.

Das Root-Element heißt *beans*Die in der Konfiguration definierten Objekte werden Spring-Beans bezeichnet. Jede Spring-Bean-Definition erfolgt in einem *bean*-Element.

Mit dem Attribut *id* wird ein Name festgelegt, über den das Objekt von der Factory angefordert werden kann. Das Attribut *class* definiert die Klasse, die Grundlage für das Objekt ist.

Die größte Herausforderung bei der Erstellung der Konfigurationsdatei ist die Angabe der Namensräume im Root-Element *beans*. Diese müssen exakt wie oben beschrieben angegeben werden. Ein Fehler führt zu Problemen, die erst zur Laufzeit zu schwer verständlichen Fehlermeldungen führen.

```
package beispiel01;

public class MathServiceImpl implements MathService {

    @Override
    public int add(int summand1, int summand2) {
      return summand1+summand2;
    }

    @Override
```

```
    public int multiply(int faktor1, int faktor2) {
      return faktor1 * faktor2;
    }
}
```
Listing 2: Erstes Beispiel für eine Spring Bean

Im Listing 2 wird die Spring-Bean mit dem Namen *mathService* implementier. Einen Hinweis das Spring Framework ist nicht zu sehen.

Auch das Interface *MathService* hat keine Beziehung zum Spring Framework.

```
package beispiel01;

public interface MathService {
    int add(int summand1, int summand2);
    int multiply(int faktor1, int faktor2);
}
```
Listing 3: Erstes Beispiel für das Interface einer Spring Bean

Im folgenden Listing wird die Verwendung der Factory demonstriert.

```
package beispiel01;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main{

    public static void main(String[] args) {
      ApplicationContext applicationContext =
            new ClassPathXmlApplicationContext(
                  "beispiel01/applicationContext01.xml");
      MathService service =
      (MathService) applicationContext.getBean("mathService");
      System.out.println(" 1 + 2 ergibt: " + service.add(1, 2));
      System.out.println(" 3 * 4 ergibt: " + service.multiply(3,4));
    }

}
```
Listing 4: Erstes Beispiel für die Verwendung der Spring Factory

Die Spring Factory wird – genauso wie die Konfigurationsdatei - *ApplicationContext* genannt. Für die Erstellung der Factory wird die Konfigurationsdatei angegeben. Diese soll im Klassenpfad im Paket *beispiel01* gesucht werden.

Die Bean wird über die Methode *getBean* geladen. Dabei wird der Name der Bean angegeben, der als Attribut id (Listing 1) definiert wurde.

Wesentliches Konzept des Spring Frameworks ist es, dass die Entwicklung nicht durch das Framework eingeschränkt wird. Dieses trifft im Beispiel uneingeschränkt für die Definition der Services – die ohne ihr Zutun Spring-Beans werden – zu.

Der dargestellte Weg, um aus dem ApplicationContext eine Spring Bean zu holen, verstößt gegen dieses Konzept – hier wird offensichtlich das Spring Framework verwendet.

Wir werden sehen, dass sich diese Einschränkung noch beheben lässt.

3.2 Das Leben einer Spring-Bean

Dieser Abschnitt stellt die Phasen dar, die eine Spring-Bean im Laufe ihrer Existenz durchläuft:

- **Definition**: In dieser Phase wird die Spring-Bean durch den Default-Konstruktor erzeugt:
- **Pre-Initialyzed**: In diser Phase wird die Spring-Bean initialisiert. Es werden entsprechend der Konfiguration Werte und Objekte zugewiesen. Auf diese Weise wird das Objektnetz aufgebaut.

- **Ready**: Hat eine Spring-Bean diesen Zustand erreicht, so kann sie von der Factory an Interessenten vergeben werden.
- **Destroyed**: Die Spring Factory kann das Leben einer Spring-Bean beenden. Zur Freigabe einer Spring-Bean können Aktivitäten definiert werden.

Standardmäßig werden Spring-Beans nur ein einziges Mal erzeugt – das ist für Service-Klassen sinnvoll.

In der Konfiguration (s. Listing 1) kann durch das Bean-Attribut *scope="prototype"* die Factory angewiesen werden, für jede Anfrage ein neues Objekt zu erzeugen.

Ist eine Spring-Bean einmal einem Benutzer ausgeliefert worden, hat die Factory keinen Einfluss auf die weitere Verwendung.

Aus dem Lebenszyklus einer Spring-Bean ergeben sich zunächst folgende Schlussfolgerungen.

- Eine Spring-Bean muss einen Standardkonstruktor definieren, um von der Factory erzeugt werden zu können[2].
- Eine Spring-Bean wird erst dann herausgegeben, wenn sie alle konfigurierten Eigenschaften besitzt. Sie ist damit fertig zum Gebrauch.

[2] Alternativ können auch parametrisierte Konstruktoren verwendet werden (s. Abschnitt 4.3)

4 Das Knüpfen von Objekt-Netzen

Wie in Kapitel **Fehler! Verweisquelle konnte nicht gefunden werden.** dargestellt, werden Services modular aufgebaut und sind daher voneinander abhängig. Aufgabe einer Factory ist es, diese Abhängigkeiten zu befriedigen.

Die Konfiguration eines Objekt-Netzes über die Konfigurationsdatei *applicationContext.xml* ist Gegenstand dieses Kapitels[3].

4.1 Attribute über Setter injizieren

Wurde ein Objekt durch die Spring-Factory erzeugt so befindet es sich im Zustand *pre-initialyzed*. Die Factory wird entsprechend der Konfiguration die Setter-Methoden des Objekts nutzen, um das Objekt zu instrumentieren.

Die Vorgehensweise wird an folgendem Beispiel demonstriert:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
  <bean id="helloService" class="beispiel04.HelloServiceImpl" >
    <property name="text" value="Hallo!" />
  </bean>
</beans>
```

Listing 5: Zuweisung eines Attributs an eine Bean

In Listing 5 wird die Spring-Bean *helloService* definiert. Mit Hilfe des Elements *property* wird das Attribut *text* des Objekts definiert. Bevor die Spring-Bean das erste Mal herausgegeben wird, wird dessen Methode *setText* aufgerufen und der angegeben Wert übergeben.

```
package beispiel04;

public class HelloServiceImpl implements HelloService {
    String text;

    public String getText() {
      return text;
    }

    public void setText(String text) {
      this.text = text;
    }

    @Override
    public String greeting() {
      return text;
    }

}
```

Listing 6: HelloServiceImpl als Gegenstand der Attribut-Injection

Einzige Voraussetzung an die Klasse der Spring-Bean ist, dass die entsprechende öffentliche Setter-Methode existiert (s. Listing 6).

Die Zuweisung von Attributen mit einem primitiven Datentyp ist ohne umstände möglich. Die Spring Factory wandelt den String in einen *int*, *double* oder *char* um.

4.1.1 Abgekürzte Schreibweise

Oftmals haben Spring-Beans eine ganze Reihe von Eigenschaften. Die Schreibweise mit property-Elementen wirkt dann eher geschwätzig.

[3] Nicht alle syntaktische Möglichkeiten werden gezeigt. Hierzu wird auf [2] verwiesen.

Kürzer ist die folgende Schreibweise

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

  <bean id="helloService" class="beispiel04.HelloServiceImpl"
    p:text="Hallo Stefan Koch" />
</beans>
```

Listing 7: Alternative, abgekürzte Schreibweise für die Setter-Injection

Die Attribute der Bean werden als Attribute des Bean-Elements angegeben. Dabei wird diesen Attributen der Namensraum-Präfix *p* vorangestellt. Der Namensraum-Präfix kann individuell gewählt werden; wesentlich ist der Verweis auf den Namensraum *http://www.springframework.org/schema/p*.

4.2 Auswertung von Property-Dateien

Wenn Sie sich vorstellen, dass Sie eine Bean haben, die eine Datenbank-Verbindung nutzt, ist es plausibel, dass die Zugangsdaten nicht in einer Spring-Konfigurationsdatei enthalten sein sollten. Es ist sinnvoll, solche Daten in separaten Property-Dateien unterzubringen. Demonstriert wird diese Möglichkeit mit dem Attribut *grussformel*, das in der Datei *gruss.properties* definiert wird.

```
grussformel=Hallo Stefan Koch!
```

Listing 8: Inhalt der Property-Datei gruss.properties

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-
3.0.xsd">

    <context:property-placeholder location=
      "classpath:beispiel04/gruss.properties"/>
  <bean id="helloService" class="beispiel04.HelloServiceImpl" >
    <property name="text" value="${grussformel}" />
  </bean>
</beans>
```

Listing 9: Konfigurationsdatei wertet die Propery-Datei aus

Um Daten einer Property-Datei auswerten zu können wird der Namensraum *context* (*http://www.springframework.org/schema/context*) importiert.

Aus diesem Namensraum wird das Element *property-placeholder* verwendet, um Property-Dateien einzubinden. Über das Attribut *location* wird der Name der Property-Datei angegeben. Im obigen Beispiel wird durch *classpath* festgelegt, dass die Angabe relativ zum Klassenpfad verstanden wird.

Die so eingelesenen Properties können über die Syntax *${property-Name}* in der Konfigurationsdatei verwendet werden.

Im Beispiel wird die Eigenschaft *text* der Spring-Bean *helloService* mit der Property *grussformel* definiert.

4.3 Spring-Beans mit parametrisierten Konstruktoren erzeugen

Neben der oben dargestellten Setter-Injection gibt es auch die Möglichkeit, ein Spring-Bean mit einem parametrisierten Konstruktor zu erzeugen.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <bean id="helloService" class="beispiel04.HelloServiceImpl" >
      <constructor-arg value="Hallo Stefan Koch" />
    </bean>
</beans>
```

Listing 10: Konfigurationsdatei mit Konstruktor-Injection

Anstelle des *property*-Elements wird *construcotr-arg* verwendet. Als *value* wird das Argument für den Konstruktor-Aufruf angegeben.

```
    public HelloServiceImpl(String text) {
      this.text = text;
    }
```

Listing 11: Passender Konstruktor zur Konstrukor-Injection

4.3.1 Passenden Konstruktor auswählen

Wenn es eine Auswahl von Konstruktoren gibt, so kann es erforderlich sein, den Datentyp des Konstruktor-Arguments anzugeben, damit die Spring-Factory den richtigen Konstruktor ausgibt.

```
<bean id="exampleBean" class="examples.ExampleBean">
<constructor-arg type="int" value="7500000"/>
<constructor-arg type="java.lang.String" value="42"/>
```

Listing 12: Konstruktor-Injection mit Typ-Angabe (aus [2])

Der Datentyp wird durch das Attribut *type* angegeben. Primitive Variablen werden als *char*, *int*, *long*, *float* und *double* angegeben. Ein String muss vollqualifiziert als *java.lang.String* angegeben werden.

Sollte die Angabe des Typs nicht ausreichen, um den passenden Konstruktor auszuwählen, so kann dem *constructor-arg* Element auch noch das Attribut *index* angegeben werden.

Mit *index* wird der Index des Parameters festgelegt.

Als weitere Möglichkeit zur Konstruktor-Auswahl steht das Attribut *name* zur Verfügung. Dieses nimmt Bezug auf den Parameter-Namen des Konstruktors.

4.4 Injektion von Objekten

Objektnetze entstehen dadurch, dass ein Objekt eine Ansammlung anderer Objekte beinhaltet. Service-Objekte nutzen oftmals andere Service-Objekte, um ihre Aufgabe zu erfüllen.

Spring-Beans lassen sich anderen Spring-Beans zuweisen.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

  <bean id="grussformel" class="java.lang.String">
    <constructor-arg value="Hallo Stefan Koch" />
  </bean>

  <bean id="helloService" class="beispiel04.HelloServiceImpl" >
    <property name="text" ref="grussformel" />
  </bean>
</beans>
```

Listing 13: Die Spring-Bean *grussformel* wird der Spring-Bean *helloService* als *text* zugewiesen

In der Konfiguration (Listing 13) wird zunächst die Spring-Bean *grussformel* erzeugt. Diese ist vom Typ *String*. Der Inhalt dieser Bean wird durch das Element *constructor-arg* festgelegt.

Die Bean *grussformel* wird der Bean *helloService* injiziert. In dem *ref*-Attribut des *property*-Elements wird dazu der Bean-Name *grussformel* angegeben.

4.5 Collections als Parameter übergeben

Die folgenden Beispiele zeigen, wie Collections als Parameter definiert werden können.

```
<bean id="moreComplexObject" class="example.ComplexObject">
    <!-- results in a setAdminEmails(java.util.Properties) call -->
    <property name="adminEmails">
       <props>
             <prop key="administrator">administrator@example.org</prop>
             <prop key="support">support@example.org</prop>
             <prop key="development">development@example.org</prop>
       </props>
    </property>
```

Listing 14: Injektion von Properties (Beispiel aus [2])

Properties werden durch das Element *props* definiert. Die Property-Einträge werden durch *prop*-Elemente definiert. Mit den Attributen *key* und *value* werden Schlüssel und Wert angegeben. Der Wert kann als Textinhalt des *prop*-Elements angegeben werden.

```

    <!-- results in a setSomeList(java.util.List) call -->
    <property name="someList">
       <list>
             <value>a list element followed by a reference</value>
             <ref bean="myDataSource" />
       </list>
    </property>
```

Listing 15: Injektion einer Liste (Beispiel aus [2])

Listen werden in der Konfigurationsdatei durch das Element *list* definiert. Listeneinträge können in *value*-Elementen angegeben werden. Listenelemente können auch mit Spring-Beans gefüllt werden. In diesem Fall wird ein *ref*-Element verwendet.

```
    <!-- results in a setSomeMap(java.util.Map) call -->
    <property name="someMap">
       <map>
             <entry key="an entry" value="just some string"/>
             <entry key ="a ref" value-ref="myDataSource"/>
       </map>
    </property>
```

Listing 16: Injektion einer Map (Beispiel aus [2])

Properties werden durch das Element *map* definiert. Die Einträge werden durch *entry*-Elemente definiert. Mit den Attributen *key* und *value* werden Schlüssel und Wert angegeben. Der Wert kann als Textinhalt des *prop*-Elements angegeben werden.

Soll der Inhalt eines Map-Eintrags eine Spring-Bean sein, so wird anstelle von *value* das Attribut *value-ref* verwendet und der Name der Spring-Bean angegeben.

```
    <!-- results in a setSomeSet(java.util.Set) call -->
    <property name="someSet">
       <set>
             <value>just some string</value>
             <ref bean="myDataSource" />
       </set>
    </property>
</bean>
```

Listing 17: Injektion eines sets (Beispiel aus [2])

Sets werden genauso wie Listen übergeben: Das Element set definiert den Inhalt des Sets. Die einzelnen Einträge werden durch *value*-Elemente definiert. Spring-Beans werden durch *ref*-Elemente angegeben.

5 Spring-Factory im Einsatz

Der in den Beispielen von Abschnitt 4 dargestellte Weg, die Spring-Factory zu nutzen, kann erheblich erleichtert werden.

- Der Nutzer eines Services sollte keine Abhängigkeit zu der konkreten Service-Klasse haben. Zur Entkopplung der Service-Klasse von dessen Nutzer wird ein Service-Interface definiert. Auf diese Weise ist es möglich, die konkrete Klasse auszutauschen, ohne dass die Nutzer davon syntaktisch betroffen sind.
- Es sollte ausreichen, dass der Nutzer der Service-Klasse den Namen des Interface kennt. Mit dieser Information sollte die Spring-Bean angefordert werden.
- Der Nutzer einer Service-Bean sollte keine direkte Abhängigkeit zur Spring-Factory haben.

Um diese Ziele zu erreichen ist für jeden Service ein Interface zu erstellen. Die Service-Klassen müssen dieses Interface implementieren.

```
package beispiel04;

public interface HelloService {
    String greeting();
}
```

Listing 18: Service Interface des HelloService

Da das Interface der sichtbare Bestandteil des Services ist, wird für das Interface der Service-Name verwendet.

In dem Beispiel befindet sich das Interface – genauso wie die Implementierung – in demselben Paket. In einer Anwendung sollten Interface und Implementierung in unterschiedlichen Paketen definiert werden.

```
package beispiel04;

public class HelloServiceImpl implements HelloService {
    private String text;

    public HelloServiceImpl(){}

    public HelloServiceImpl(String text) {
       this.text = text;
    }

    public String getText() {
       return text;
    }

    public void setText(String text) {
       this.text = text;
    }

    @Override
    public String greeting() {
       return text ;
    }
}
```

Listing 19: Service-Implemtierung

Die Service-Klasse muss das Service-Interface implementieren.

Die Implementierung des Service wird oft dadurch kenntlich gemacht, dass dem Service-Namen die Endung ‚Impl' folgt. *HelloServiceImpl* implementiert das Interfache *HelloService*.

Der *ServiceLocator* übernimmt die Aufgabe der Factory: Auch er wird zunächst als Interface definiert.

```
package beispiel04;

public interface ServiceLocator {
```

```
    HelloService getHelloService();
}
```
Listing 20: Definition des ServiceLocator-Interfaces

Der *ServiceLocator* definiert Zugriffsmethoden für jede Service-Klasse. Zu beachten ist der Rückgabe-Typ der *get*-Methode: hier wird das Service-Interface verwendet.

Im wird für den *HelloService* die Methode *getHelloService* deklariert.

Die Implementierung des *ServiceLocator*-Interfaces wird durch das Spring Framework vorgenommen.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-3.0.xsd"
>
    <bean id="serviceLocator" class=
      "org.springframework.beans.factory.config.ServiceLocatorFactoryBean">
      <property name="serviceLocatorInterface"
            value="beispiel04.ServiceLocator" />
    </bean>

    <bean id="serviceLocatorFactory"
      class="beispiel04.ServiceLocatorFactory" >
            <property name="serviceLocator" ref="serviceLocator" />
    </bean>
</beans>
```
Listing 21: Konfiguration der Spring-Bean serviceLocatorFactory

Die *ServiceLocatorFactoryBean* von Spring übernimmt die Aufgabe der Factory. Die Methoden der Factory sind durch das übergebene Interface *ServiceLocator* definiert. Intern wird der Rückgabe-Typ der Methoden verwendet, um die richtige Spring-Bean zurückzuliefern.

Gibt im Beispiel (Listing 20) die Methode *getHelloService* ein Objekt vom Typ *HelloService* zurück, so sucht die *ServiceLocatorFactoryBean* nach einer SpringBean von diesem Typ.

Im letzten Schritt ist noch eine Factory für die Factory (= Abstract Factory) zu realiseren. Diese wird als *ServiceLocatorFactory* bezeichnet. Die *ServiceLocatorFactory* muss der Entwickler selbst schreiben: Hierin wird festgelegt, welche Konfiguration für die Erstellung der Spring-Beans verwendet wird. Besonders elegant: sie bekommt die ServiceLocatorFactoryBean durch den Spring-Container injiziert (s. Listing 21).

```
package beispiel04;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class ServiceLocatorFactory {

    private static ServiceLocator serviceLocator = null;

    public static ServiceLocator getServiceLocator() {
      if (serviceLocator == null) {
            new ClassPathXmlApplicationContext(
                        new String[] {
                                    "beispiel04/applicationContext04.4.xml"
                                    ,"beispiel04/ServiceLocator.xml"
                        });
      }
      return serviceLocator;
    }
```

```
    public void setServiceLocator(ServiceLocator serviceLocator) {
      ServiceLocatorFactory.serviceLocator = serviceLocator;
    }
}
```
Listing 22: Definition einer ServiceLocatorFactory

Die statische Methode *getServiceLocator* stellt die Factory-Methode dar. Sie liefert dem Nutzer den *ServiceLocator* (eigentlich die *ServiceLocatorFactoryBean*) zurück. Ist der ServiceLocator nicht definiert, so wird der *ApplicationContext* initialisiert.

Bei der Initialisierung (s. Listing 22) werden zwei Konfigurationsdateien verwendet. Die erste enthält die fachlich erforderlichen Beans, die zweite dient ausschließlich der Definition des *ServiceLocators* und der *ServiceLocatorFactory*. .

Die Nutzung des ServiceLocators ist vollständig transparent: Es ist nicht ersichtlich, dass für die Factory das Spring-Framework eingesetzt wird.

```
package beispiel04;

public class Main04 {
    public static void main(String[] args) {
      HelloService hello =
            ServiceLocatorFactory.getServiceLocator().getHelloService();
      System.out.println(hello.greeting());
    }
}
```
Listing 23: Einsatz der ServiceLocatorFactory

5.1 Die ServiceLocatorFactory im Überblick

Das folgende Klassendiagramm liefert einen Überblick über den Aufbau und die Funktionsweise der *ServiceLocatorFactory*.

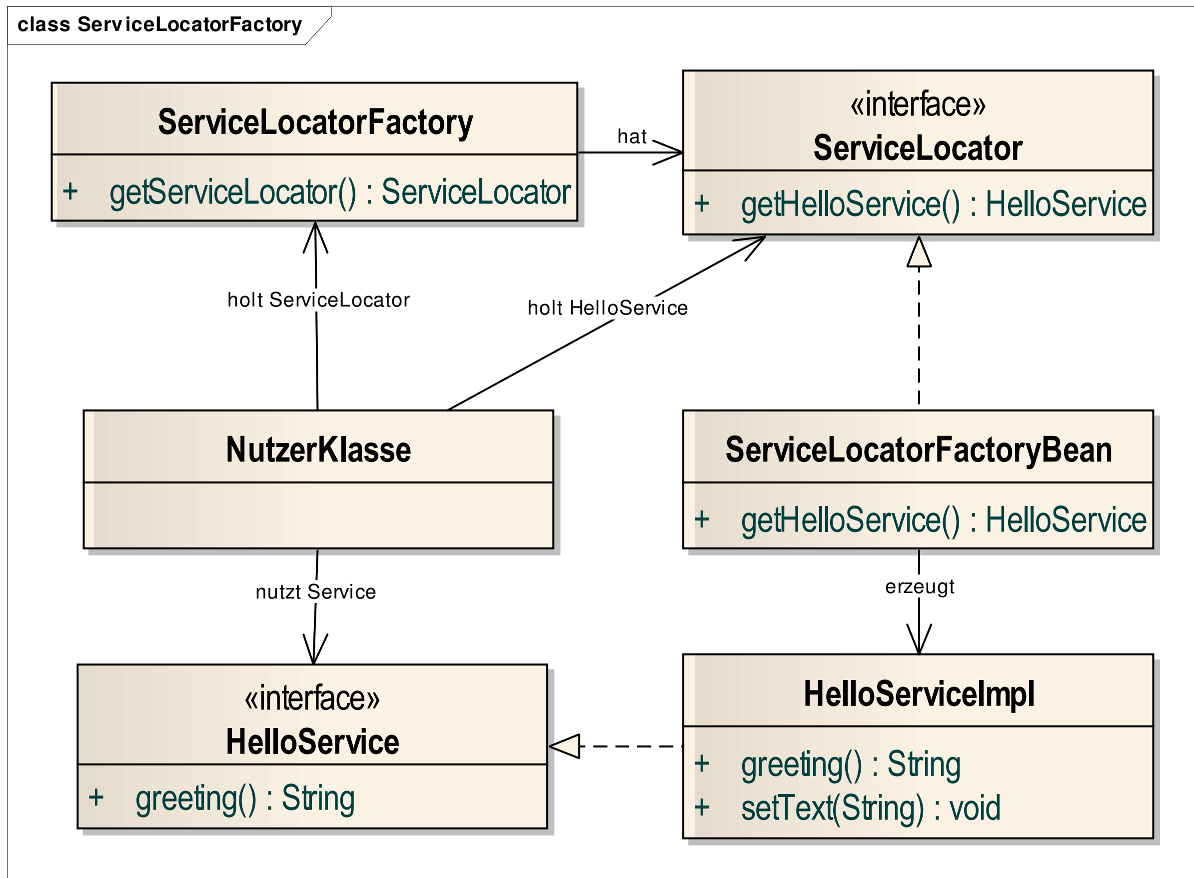


Abbildung 5: Klassendiagramm zur ServiceLocatorFactory

Die *ServiceLocatorFactory* stellt aus Perspektive eines Nutzers (*NutzerKlasse*) die Factory für die gewünschte Service-Factory dar.

Der Nutzer wird sich im ersten Schritt über die Methode *getServiceLocator* einen *ServiceLocator* holen. Im zweiten Schritt wird der Nutzer mit der Methode *getHelloService* der *ServiceLocator*-Instanz den Service *HelloService* ermitteln.

6 Aspektorientierter Ansatz

Stellen Sie sich vor, sie möchten Häufigkeit und Ausführungszeiten aller Services ermitteln, um Performance-Messungen durchzuführen. Eine derartige Anforderung lässt sich dadurch erfüllen, dass die Performance-Messung in jeden einzelnen Service implementiert wird.

Mit einem aspektorientierte Ansatz wird dieser Aspekt des Profilings einmalig implementiert und bei Bedarf für unterschiedliche Services angewandt.

Der Vorteil liegt auf der Hand: Aspekte lassen sich getrennt von der Geschäftslogik entwickeln. Ein Grundprinzip guter Programmierung ‚Don't repeat yourself' (s. [7]) bleibt gewahrt.

Der aspektorientierte Ansatz hat – möglicherweise unbemerkt - Einzug gehalten: Das Transaktions-Management für Enterprise Java Beans ist das bekannteste Beispiel.

Spring bietet die Möglichkeit, Aspekte für alle Spring-Beans vorzusehen.

6.1 Fachbegriffe der aspektorientierten Programmierung

Um über aspektorientierte Programmierung sprechen zu können, wird die Bedeutung der Fachbegriffe geklärt, die für die aspektorientierte Programmierung mit dem Spring-Framework eine Rolle spielen.

6.1.1 Aspekt

Der Aspekt beinhaltet das Thema, das durch die aspektorientierte Programmierung in Angriff genommen wird: Themen könnten sein Transaktionsmanagement, Autorisierung oder Protokollierung.

6.1.2 Joinpoint

Der Joinpoint ist der Punkt im Programmfluss, an dem der Aspekt berücksichtigt wird. Bei der Implementierung mit Spring kann der Joinpoint nur bei einem Methodenaufruf liegen. Innerhalb einer Methode lässt sich kein Joinpoint definieren.

6.1.3 Advice

Der Advice stellt die Implementierung eines Aspekts dar: Ist beispielsweise der Aspekt die Protokollierung, so wird im Advice definiert, was und wie protokolliert wird.

Pointcut

Als Pointcut wird die Menge der Joinpoints verstanden. Ein Pointcut wird durch Regeln definiert. Eine Regel könnte sein: berücksichtige alle set-Methoden aus einem angegebenen Paket.

Weaving

Unter Weaving (engl. weben) wird die Einbindung des Advices an den Joinpoints verstanden.

Advisor

Der Adviser sorgt dafür, dass der Advice am Joinpoint ausgeführt wird.

6.2 Prinz der Aspektorientierten Programmierung mit dem Spring-Framework

Aspektorientierte Programmierung wird mit dem Spring-Framework durch die Verwendung von Proxies implementiert.

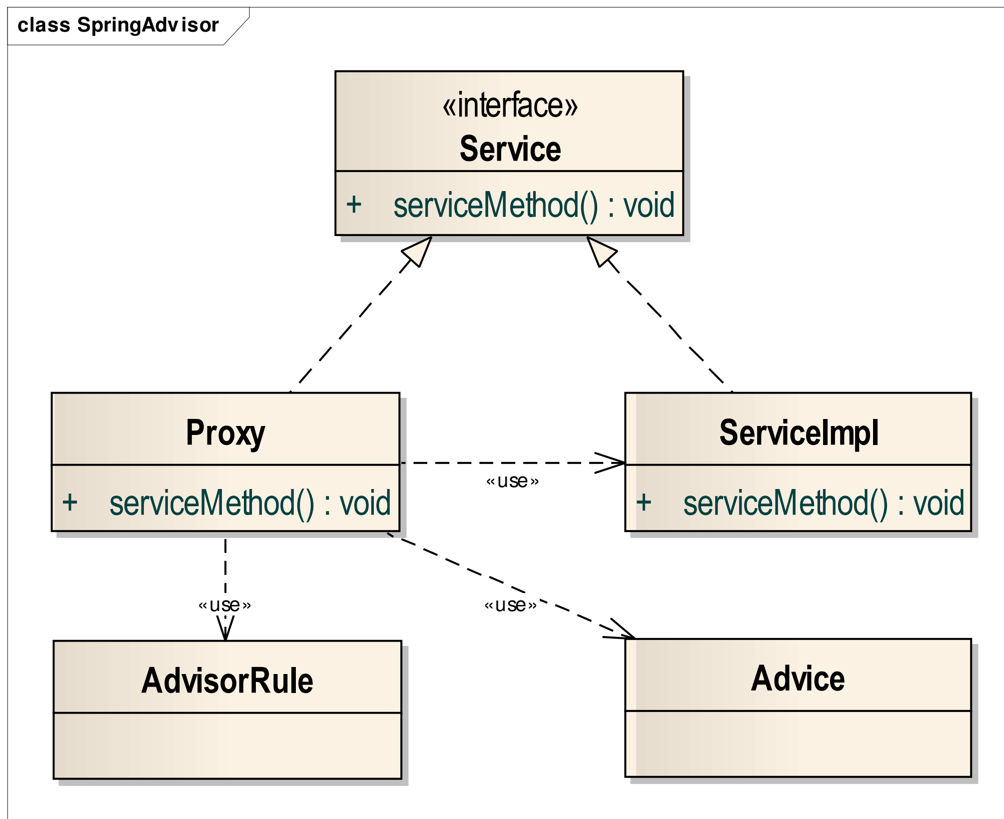


Abbildung 6: Proxy-Ansatz zur Implementierung des Aspektorientierten Ansatzes

Eine Proxy-Klasse, die das Service-Interface implementiert, wird dem Nutzer zur Verfügung gestellt. Wird eine Operation des Proxies aufgerufen, so verwendet der Proxy die Klasse *ServiceImpl*, damit die fachliche Logik ausgeführt wird.

Ob zusätzlich zur fachlichen Ausführung noch ein Aspekt berücksichtigt werden muss, wird durch Advisor-Rules entschieden: Eine Regel könnten lauten: führe einen bestimmen Advice vor dem Aufruf der Methoden aus, deren Name mit *set* anfängt.

In diesem Fall benutzt der Proxy eine Methode der Advice-Klasse, um diese vor dem Methoden-Aufruf der ServiceImpl-Methode auszuführen. Ein Advice, der vor dem Methoden-Aufruf angewandt wird, heißt Before-Advice.

Das folgende Sequenz-Diagramm zeigt den Aufruf der Service-Methode *serviceMethod*, wenn ein Aspekt durch einen Before-Advice berücksichtigt werden soll.

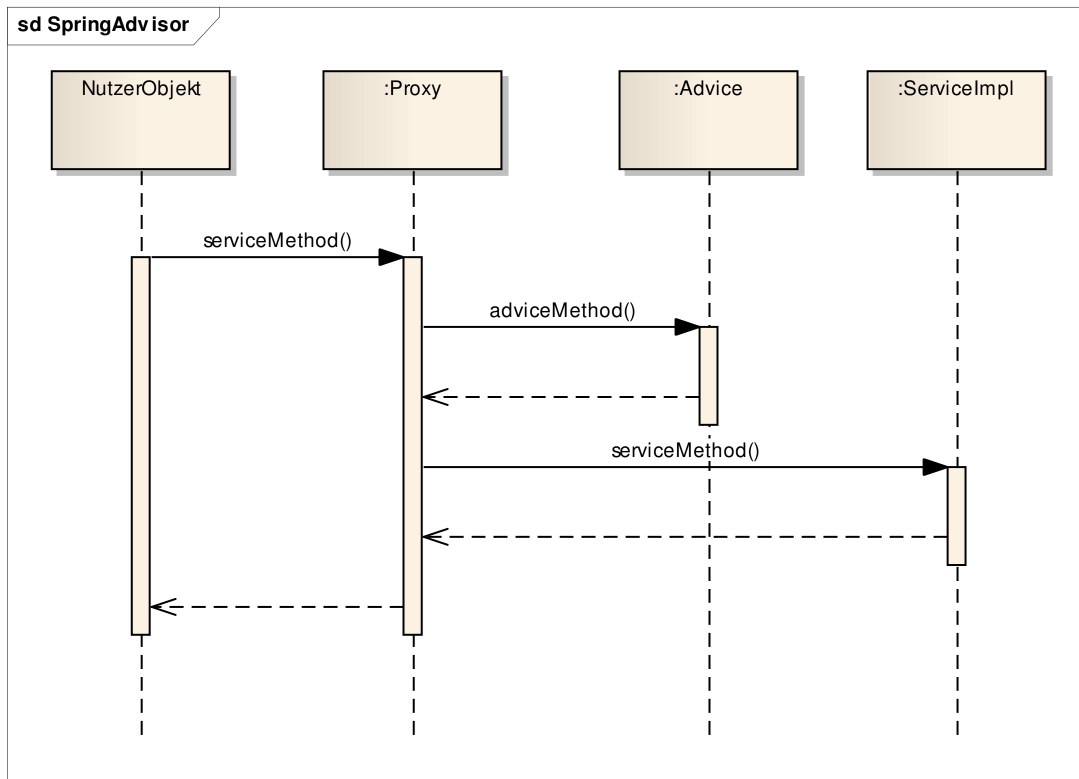


Abbildung 7: Einsatz eines Before-Advice

Der Nutzer verwendet einen Proxy anstelle der Service-Klasse. Beim Aufruf der *serviceMethod* ruft der Proxy zunächst die *adviceMethod* der Klasse Advice auf, bevor *serviceMethod* der *ServiceImpl*-Klasse aufgerufen wird.

Beim Einsatz der Spring-Frameworks trifft der Nutzer keine bewußte Entscheidung zur Nutzung eines Proxies. Er benutzt die Spring-Bean, die ihm von der Factory geliefert wird. Die Factory entscheidet somit, für welche Spring-Bean eine Proxy-Klasse einzusetzen ist.

Für den Einsatz der aspektorientierten Programmierung werden bei Spring benötigt:

- Ein Proxy: dieser wird durch Spring das Spring-Framework erzeugt,
- ein Advice: der Advice wird durch den Entwickler beigesteuert,
- AdvisorRules: diese werden durch Konfiguration definiert,

6.3 Konfiguration der Aspektorientierten Programmierung

Die Konfiguration des Advisors durch Advice und Pointcut kann über die Konfigurations-Datei von Spring, der *applicationContext.xml*, erfolgen.

6.3.1 Advice-Interfaces

Ein Advice kann durch Implementierung spezieller Advice-Interfaces implementiert werden. Der Typ des Interfaces entscheidet über die Art des Advices.

Interface	Methode	Verwendung
AfterReturningAdvice	afterReturning	Wird nach der Operation aufgerufen.
MethodBeforeAdvice	before	Wird vor der Operation aufgerufen. Eignet sich beispielsweise für die Autorisierung von Zugriffen
MethodInterceptor	invoke	Ist ein Around-Advice: Der Aspekt kann vor und nach der Operation zur Ausführung kommen. Beispiel: Transaktions-Management.
ThrowsAdvice	afterThrowing	Wird zum Abfangen bestimmter Exceptions verwendet. Beispiel: Exception-Translation.

Tabelle 1: Überblick häufig verwendeter Advice-Interfaces

6.3.2 Interface MethodInterceptor

Am Beispiel des Interface MethodInterceptor wird die Erstellung von Advices beispielhaft erläutert. Der MethodInterceptor-Advices sind besonders mächtig, da sie sowohl vor, als auch nach wie auch beim Auftreten von Exceptions Aspekte berücksichtigen können.

```
import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

public class MyMethodInterceptor implements MethodInterceptor {
    private static final Log log =
      LogFactory.getLog(MyMethodInterceptor.class);
    @Override
    public Object invoke(MethodInvocation invocation) throws Throwable {
      log.debug("call invocation ");
      Object returnValue = invocation.proceed();
      log.debug("Method " + invocation.getMethod()
      + " has been executed.");
      return returnValue;
    }
}
```

Listing 24: Beispiel für den Einsatz des MethodInterceptor-Interfaces

Der Advice erfüllt die Aufgabe, dass vor und nach dem Methodenaufruf eine Nachricht in das Protokoll geschrieben wird.

Die Logik des Aspekts wird in der Methode *invoke* implementiert. Die Geschäfts-Methode wird durch die Methode *proceed* der Klasse *MethodInvocation* ausgeführt. Wesentlich ist, dass der Rückgabewert der Geschäftsmethode (*returnValue*) auch als Rückgabewert der Advice-Methode verwendet wird.

m vorgegebenen Fall des Listing 24 wird vor dem Aufruf der Operation die Nachricht *call invocation* ausgegeben. Die Operation wird durch die Methode *proceed* ausgeführt. Die Argumente für den Methodenaufruf sind in dem *MethodInvocation*-Objekt hinterlegt. Das Ergebnis wird der Variablen *returnValue* zugewiesen.

Nachdem die Operation ausgeführt wurde, wird erneut protokolliert. In diesem Fall wird mit Hilfe des *MethodInvocation*-Objekts der Methoden Name mit ausgegeben

6.3.3 Advisor-Konfiguration

In der Konfigurationsdatei wird ab der Version 2.5 des Spring-Frameworks das Konfigurations-Element *aop:config* verwendet, um einen Advisor zu konfigurieren.

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:aop="http://www.springframework.org/schema/aop"
      xsi:schemaLocation="
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
            http://www.springframework.org/schema/aop
            http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">

   <bean id="myAdvice" class="beispiel06.MyMethodInterceptor"/>

   <aop:config>
      <aop:pointcut id="myPointcut"
         expression="execution(* beispiel06.Service*.*(..))" />
      <aop:advisor advice-ref="myAdvice" pointcut-ref="myPointcut" />
   </aop:config>

   <bean id="service" class="beispiel06.ServiceImpl" />
</beans>
```

Listing 25: Konfiguration eines Advisors in der Konfigurationsdatei

In dem Beispiel wird das Advice durch die Klasse *MyMethodInterceptor* definiert. Das Advice wird als Spring-Bean unter dem Namen *myAdvice* konfiguriert.

Das Element *aop:config* enthält alle übrigen erforderlichen Zutaten für den Advisor. Als erstes wird definiert, für welche Methoden der Advice verwendet werden soll: *aop:pointcut*. Das Attribut *id* vergibt einen Namen für den Pointcut, das *expression*-Attribut enthält das Muster, um die Methoden festzulegen.

Verwendet wird hier die AspectJ-Notation: Nach dem Schlüsselwort *execution* wird ein Muster für Paket, Klasse und Methoden-Namen angegeben. Der Stern ersetzt Teile dieser Spezifikation.

Das Element *aop:advisor* vereinigt Advice und Pointcut.

References / Literaturverzeichnis

[1] Springframework: http://www.springsource.org/

[2] Springframework, Reference Manual: http://www.springsource.org/

[3] Google: http://www.google.com

[4] Design Patterns: Elements of Reusable Object-Oriented Software; Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides

[5] Design Pattern Data Access Object: http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html

[6] Homepage Hibernate: http://www.hibernate.org

[7] Grundsätze der Codierung: http://www.clean-code-developer.de