

## 2 Objekte finden zueinander: Dependency Injection

### 2.1 Übersicht

In diesem Kapitel wird mit Dependency Injection eines der wesentlichen Konzepte von Spring eingeführt. Zunächst wird dazu die Beispielanwendung erläutert (Abschnitt 2.2), die im gesamten Buch verwendet wird. Da Dependency Injection ein Weg ist, um Objektnetze aufzubauen, werden anschließend die »klassischen« Möglichkeiten aufgezeigt, um solche Netze zu erzeugen (Abschnitt 2.3). Schließlich wird Dependency Injection allgemein eingeführt (Abschnitt 2.4), und der Ansatz von Spring wird näher erläutert (Abschnitt 2.5). Es folgt in Abschnitt 2.6 ein Überblick über die Vorteile von Dependency Injection. Welche Teile eines Systems mit Dependency Injection konfiguriert werden können, zeigt Abschnitt 2.7. Die automatische Erzeugung von Objektnetzen (»Autowiring«) wird in Abschnitt 2.8 erläutert. Abschnitt 2.9 stellt den `ApplicationContext` vor, der einige andere Dienste neben Dependency Injection anbietet. Dann werden fortgeschrittene Themen im Bereich Dependency Injection in Abschnitt 2.10 erläutert.

Abschnitt 2.11 beschreibt die Konfiguration von Dependency Injection mithilfe von Java Annotationen. Auch Spring-Beans selbst können durch Annotationen definiert werden (Abschnitt 2.12), und mit JSR 330 steht auch ein Standard für solche Annotationen zur Verfügung (Abschnitt 2.13). Abschnitt 2.14 stellt eine weitere Möglichkeit vor: Man kann Java-Klassen schreiben, die Spring-Beans erzeugen. Schließlich wird die Auswirkung von Dependency Injection auf das Testen als ein wesentlicher Vorteil dargestellt (Abschnitt 2.15). Die Abwägung, wann man Annotationen nutzen sollte, wird in Abschnitt 2.16 vorgenommen.

## 2.2 Die Beispielanwendung

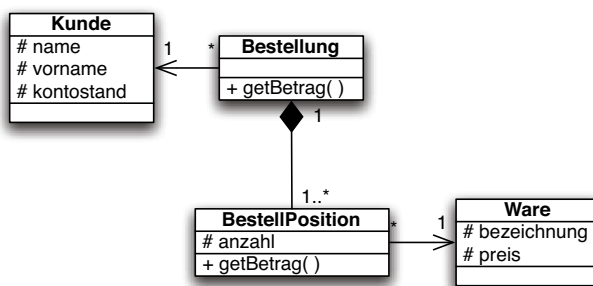
Um die in diesem Buch erläuterten Konzepte gleich in einer praxisnahen Verwendung zu sehen, wird im gesamten Buch ein durchgängiges Beispiel verwendet. Da die meisten individuellen Softwareprojekte im Bereich der Geschäftsanwendungen liegen, steht eine solche Anwendung auch hier im Mittelpunkt. Man kann natürlich nicht erwarten, dass dieses Beispiel den etablierten ERP- oder CRM-Anwendungen das Wasser reichen kann. Daher wird die Anwendung nur eine einfache Version dessen implementieren, was man in einer Geschäftsanwendung vorfindet.

Konkret soll es mit dieser Anwendung möglich sein, Bestellungen zu bearbeiten. Dazu muss zum einen der Bestellprozess modelliert werden und zum anderen die davon betroffenen Business-Objekte, nämlich der Kunde, die Waren und schließlich die Bestellung selbst.

### 2.2.1 Das fachliche Modell

Das fachliche Modell zeigt Abbildung 2–1. Es gibt die Klasse Kunde mit den Attributen name, vorname und kontostand. Diese Attribute sind in den Implementierungsklassen durch private-Attribute umgesetzt, die public-Zugriffsmethoden haben. Ein weiteres Business-Objekt ist die Bestellung, die eine Komposition aus BestellPositionen ist. Diese haben jeweils Referenzen auf die Waren.

**Abb. 2–1**  
 Fachliches  
 Anwendungsmodell:  
 Die fachlichen Klassen  
 Kunden, Waren und  
 Bestellungen



Die fachlichen Klassen haben kaum eigene Logik. Lediglich die Berechnung des Betrags der Bestellung ist hier implementiert. Solche Entwürfe findet man in der Praxis recht häufig: Die Business-Objekt-Schicht dient nur zum Verwalten der Daten und hat wenig echte Logik. Die meiste Logik liegt auf der Ebene der Prozesse und kann daher in den fachlichen Klassen nicht abgebildet werden. Dieser Ansatz wird allerdings auch kritisiert [Fow02] [Eva03], weil man zwar Business-Objekte implementiert, aber die Vorteile von Objektorientierung wie Polymorphie und Vererbung nicht nutzt.

Die Business-Objekte sollen natürlich dauerhaft in einer Datenbank gespeichert werden. Dazu bekommen die einzelnen Objekte jeweils eine *id* als Ganzzahl, die als eindeutiger technischer Schlüssel dient.

Ein klassisches Problem bei der Entwicklung von Geschäftsanwendungen ist die Frage, wie man die Objekte in der Datenbank ablegt. Die Datenbanken erzwingen meist ein relationales Modell: Die Daten müssen in Tabellen gespeichert werden. Einzelne Datensätze haben einen Wert, der sie eindeutig identifiziert (der *Primärschlüssel*). Referenzen zwischen Objekten müssen also durch Beziehungen zu Primärschlüsseln (sogenannte *Fremdschlüssel*) ersetzt werden.

Zur Abbildung der Objekte auf die Datenbankstrukturen wird in der Beispielanwendung das Pattern DATA ACCESS OBJECT (DAO) verwendet. Dies wird im Abschnitt 5.2 ausführlich dargestellt. Hier reicht der Hinweis, dass die DAOs den Zugriff auf die Datenbank vollständig kapseln. Sie erlauben es, einzelne Business-Objekte aus der Datenbank zu lesen, sie dort zu erzeugen und zu aktualisieren sowie Listen von Business-Objekten über geeignete Zugriffsmethoden (Abfragen) bereitzustellen. Einige Technologien für die Implementierung der DAOs zeigt Kapitel 5.

Datenbankzugriff

Data Access Object (DAO)

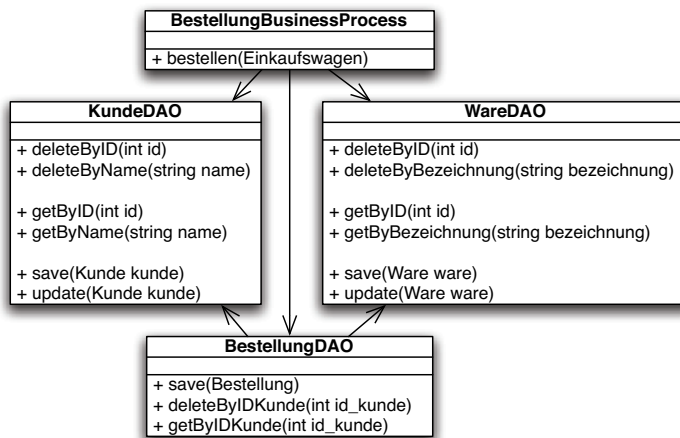


Abb. 2-2

Die DAOs in der  
Beispielanwendung

Die DAOs (Abb. 2-2) sollen möglichst keine Auswirkungen auf die Implementierung der Business-Objekte haben. Die verwalteten Objekte müssen jeweils ein Attribut *id* für den Primärschlüssel haben, aber darüber hinaus nimmt die Implementierung keine Rücksicht auf die Persistenz. Insbesondere werden beispielsweise Bestellungen mit Referenzen auf die Kunden und die BestellPositionen ausgestattet. Die Fremdschlüsselbeziehungen werden also in den DAOs aufgelöst. Entsprechend hat das DAO für die Bestellungen auch eine Referenz auf

das DAO für die Kunden, um gegebenenfalls Kunden laden oder speichern zu können.

Durch die DAOs sind auch gleich Funktionalitäten implementiert, um neue Kunden oder Waren anzulegen und sie zu verwalten.

### 2.2.2 Geschäftsprozesse in der Beispielanwendung

Die Beispielanwendung soll als Geschäftsprozess einen der wichtigsten Prozesse jedes Unternehmens abbilden: die Bestellung («Kunde droht mit Auftrag»). Die Schnittstelle des Geschäftsprozesses (Abb. 2–2) verwendet den Einkaufswagen. Er enthält Informationen über den Kunden und die bestellten Waren. Dabei werden nur primitive Datentypen verwendet, wie dies bei serviceorientierten Architekturen oder verteilten Systemen oft der Fall ist. So kann man den Dienst verwenden, ohne viel über die dahinter liegenden Business-Objekt-Strukturen zu wissen. Die bestellten Waren und der bestellende Kunde wird also nur durch seine `id` identifiziert. Entsprechend muss der Geschäftsprozess die Business-Objekte zu diesen `ids` zusammensuchen und anschließend die Bestellung erzeugen. Dabei soll noch eine Überprüfung stattfinden, ob der Kunde mit seinem derzeitigen Kontostand die Bestellung überhaupt bezahlen kann.

Um auf die persistenten Daten zugreifen zu können, muss man natürlich den Geschäftsprozess mit Referenzen auf die DAOs ausstatten.

### 2.2.3 Benutzeroberfläche

Die Beispielanwendung kommt ohne Benutzeroberfläche aus. Stattdessen werden die Funktionalitäten durch Tests sichergestellt. Parallel zur Einführung passender Spring-Technologien für Weboberflächen werden später verschiedene Benutzeroberflächen implementiert.

## 2.3 Objektnetze in OO-Systemen

Das Klassenmodell für die DAOs und den Geschäftsprozess aus dem letzten Abschnitt stellt ein Netz von Objekten dar, die jeweils Dienste implementieren. Dabei wird keine Vererbung verwendet. Es kann zwar sein, dass in der Implementierung technisch motivierte Oberklassen z.B. für die DAOs oder auch für die Geschäftsobjekte gebildet werden, aber die konzeptionellen Diagramme enthalten keine einzige Vererbungsbeziehung.

In den meisten objektorientierten Systemen kommen solche Netze von Objekten häufiger vor als Vererbung. Tiefe Vererbungsbeziehungen können sogar ein Zeichen für schlechten Code sein. Ein solches Design ist z.B. ein häufiger Fehler von Entwicklern, die Objektorientierung neu erlernen und versuchen, alle Probleme durch Vererbung zu lösen.

Die gängigen objektorientierten Programmiersprachen bieten nur eine direkte Unterstützung für Vererbung an. Delegation und Beziehungen zwischen Objekten müssen explizit ausprogrammiert werden, während man eine Vererbungsbeziehung mit einem einfachen Schlüsselwort wie `extends` bei Java erreicht.

Hinzu kommt, dass gerade bei Java Delegation oft die einzige Möglichkeit ist, um doppelten Code zu vermeiden. Vererbung ist in Java nämlich auf eine Klasse beschränkt, bei Delegation hingegen kann es beliebig viele Delegationsziele geben.

### 2.3.1 Netze weben

Die interessante Frage ist nun, wie man ein solches Netz von Objekten erzeugt. Dazu gibt es eine breite Auswahl an Pattern, Idioms und Technologien.

Man kann z.B. den einzelnen Objekten die Referenzen von außen zuweisen. Dies könnte im Hauptprogramm geschehen. Will man z.B. dem `BestellungBusinessProcess` eine Referenz auf das `KundeDAO` zuweisen, so könnte der Code wie in Listing 2-1 gezeigt aussehen.

```
public static void main() {  
    BestellungBusinessProcess bbp =  
        new BestellungBusinessProcess();  
    bbp.setKundeDAO(new KundeDAO());  
}
```

*Referenzen von außen  
zuweisen*

#### **Listing 2-1**

*Referenzen von außen  
zuweisen*

In großen Anwendungen wird dieser Ansatz meistens nur für Teile des Objektnetzes genutzt, da sonst der Code zu unübersichtlich wird. Zum Erstellen eines sehr kleinen Objektnetzes ist dieses Vorgehen jedoch weit verbreitet, da es einfach zu implementieren ist.

Objekte können andere Objekte selbst erzeugen. So könnte in der Beispielanwendung der Bestellung-Geschäftsprozess die abhängigen DAOs selbst erzeugen und auf sie zugreifen. Diese Möglichkeit zeigt Listing 2-2.

```
public BestellungBusinessProcess() {  
    this.kundeDAO = new KundeDAO();  
}
```

*Objekte selbst erzeugen*

#### **Listing 2-2**

*Objekt im Konstruktor  
selbst erzeugen*

Auch dieses Verfahren ist recht einfach zu implementieren. Es hat allerdings den Nachteil, dass man nur durch Änderung des Codes andere Implementierungsklassen verwenden kann. Dazu muss man jedoch nicht nur *eine* Stelle ändern, sondern jede Stelle, an der Instanzen der betroffenen Klasse erzeugt werden. Also ist auch hier die Änderbarkeit und Wartbarkeit nur bedingt gegeben.

Das Factory-Pattern

Eine bekannte Lösung für dieses Problem ist das FACTORY-Pattern [GHJV94]. Dabei werden Objekte nicht direkt mit `new`, sondern durch eine statische Methode einer Klasse oder von einem dedizierten Objekt mit entsprechenden Instanzmethoden erzeugt (Listing 2–3).

**Listing 2–3**  
Erzeugung mithilfe  
einer Factory

```
public BestellungBusinessProcess() {  
    this.kundeDAO = Factory.getKundeDAO();  
}
```

Dies hat den Vorteil, dass die Erzeugung an einer zentralen Stelle im Code geschieht und damit leichter änderbar und wartbar ist. So können beispielsweise auch Subklassen der deklarierten Rückgabetypen erzeugt werden. Oder es werden an der FACTORY-Schnittstelle nur *Interfaces* deklariert, so dass irgendeine Implementierung eines solchen Interfaces zurückgegeben werden kann. Dadurch erreicht man eine bessere Entkopplung von der konkreten Implementierung, so dass das System leichter änderbar ist. Man kann dann auch Objekte auf einem Server statt lokaler Objekte verwenden. Das ist notwendig, wenn man das System sowohl mit einem Application-Server als auch in einer Java-SE-Umgebung laufen lassen will, um z.B. Tests ohne Application-Server zu ermöglichen. Für Tests haben die Factories zudem den Vorteil, dass man Mocks in das System einführen kann, also Objekte, die eine reduzierte Funktionalität für Tests implementieren. Erst dadurch kann man einzelne Teile des Codes unabhängig voneinander testen: Man kann die abhängigen Objekte durch Mocks ersetzen. In der Praxis wird wegen dieser Vorteile häufig der Ansatz der Erzeugung mithilfe einer FACTORY verwendet, da gerade die Testbarkeit eines Systems erst durch die Factories gewährleistet ist.

Es gibt aber auch Nachteile: Man muss aufpassen, dass das Pattern nicht durch Erzeugung »an der FACTORY vorbei« mit einem einfachen `new` unterlaufen wird. Dann ist die Änderung in der FACTORY nämlich nicht mehr ausreichend, um wirklich überall ein anderes Objekt zu erzeugen.

Ein weiteres Problem in der Praxis ist die Komplexität, die diese Lösung erreichen kann. Wenn man an der Schnittstelle statisch den richtigen Typ deklarieren will, führt dies zu einer Vielzahl von Methoden, durch die jeweils ein spezielles Produkt erzeugt werden kann. Will man die FACTORIES z.B. um die Möglichkeit erweitern, Mocks zurück-

zugeben, kann es gut sein, dass man einen erheblichen Aufwand hat, weil man viele FACTORIES im System hat und diese eventuell auch noch eine Vielzahl an Methoden haben.

Alternativ kann die Schnittstelle der FACTORY dynamisch typisiert werden: Man deklariert als Rückgabewert Object und wandelt in den erwarteten Typ um. Dadurch verliert man natürlich die Typsicherheit zur Kompilierungszeit.

In der Beispielanwendung könnte man eine FACTORY für die DAOs definieren. Für jedes DAO könnte die FACTORY eine Methode haben, um ein entsprechendes DAO zu erzeugen. Dadurch wäre sie statisch typisiert. Man kann dann die Persistenzschicht leicht austauschen, indem man die FACTORY so modifiziert, dass sie andere Objekte zurückgibt. Ein Mocking der DAOs für Tests müsste allerdings bei der Implementierung der FACTORY beachtet werden: Es müssen Methoden implementiert werden, mit denen man der FACTORY ein Mock »unterschieben« kann, das statt der DAOs ausgeliefert wird. Wahrscheinlich ergibt sich letztendlich ein Design, in dem man für alle FACTORIES im System eine gemeinsame Oberklasse entwickelt.

Das wesentliche Problem des FACTORY-Ansatzes ist, dass bei der Verwendung der FACTORIES die meisten Klassen im System von einer FACTORY abhängen, so dass eine Wiederverwendung einzelner Klassen oder isolierte Änderungen nicht möglich sind – man muss neben der Klasse selbst auch immer die FACTORY wieder verwenden oder ändern. Man hat also eine recht starre Struktur des Systems, aus dem man nicht einfach Teile entfernen oder ersetzen kann.

Es ist vielleicht überraschend, dass auch das SINGLETON-Pattern [GHJV94] in den Bereich des Aufbaus von Objektnetzen fällt. Eigentlich soll durch dieses Pattern nur erreicht werden, dass es genau eine Instanz einer bestimmten Klasse geben kann. Gleichzeitig wird diese Instanz aber an einer wohlbekanntem Stelle im System hinterlegt, so dass sie von überallher zugreifbar ist. Dies bedeutet, dass vom SINGLETON abhängige Klassen nicht ohne Weiteres zu erkennen sind, weil jeder Teil des Systems im Prinzip auf das SINGLETON zugreifen kann.

Wie schon bei den FACTORIES gilt: Wenn man keine Möglichkeiten für das Ersetzen oder Umkonfigurieren des SINGLETONS vorsieht, kann man den Code nur noch schwer testen oder ändern. Aber selbst wenn es solche Möglichkeiten gibt, muss man aufpassen, dass man das SINGLETON auch immer ersetzt, wenn es notwendig ist. Prinzipiell kann jede Klasse von dem SINGLETON abhängen, denn von außen ist nicht erkennbar, welche Klassen das SINGLETON wirklich verwenden.

Bei der Beispielanwendung wären die DAOs ein guter Kandidat für SINGLETONS, da eine Instanz ausreichend ist, um den Datenbankzu-

*Das Singleton-Pattern*

griff zu realisieren. Den dazu passenden Code zeigt Listing 2–4: Die Implementierung der Klasse `KundeDAO` gewährleistet, dass es von dieser Klasse nur eine Instanz geben kann. Da der Konstruktor `private` ist, kann von außen keine Instanz erzeugt werden. Die Methode `getInstance()` gibt die einzige Instanz zurück.

**Listing 2–4**  
Erzeugung mithilfe  
eines Singletons

```
public class BestellungBusinessProcess {
    public BestellungBusinessProcess() {
        this.kundeDAO = KundeDAO.getInstance();
    }
    ...
}

public KundeDAO {

    private KundeDAO() {}

    private static KundeDAO instance
        = new KundeDAO();

    public static KundeDAO getInstance() {
        return instance;
    }
    ...
}
```

Wenn man ohne Datenbank testen und daher die DAOs gegen andere Implementierungen austauschen will, ergeben sich bei dieser Implementierung Probleme, weil die Ersetzbarkeit nicht bei dem Entwurf der SINGLETONS beachtet worden ist. Es fehlt eine Methode, mit der man ein eigenes Objekt (z.B. ein Mock) in der `static`-Variable ablegen kann. Das Problem des SINGLETONS ist eng verwandt mit dem Problem der statischen Methoden: Methoden, die als `public static` definiert sind, können wie ein SINGLETON von überallher aufgerufen werden, was dieselben Nachteile bezüglich Mocking und dem Erkennen von Abhängigkeiten mit sich bringt.

#### Namenssysteme

Diese Probleme sind nicht begrenzt auf selbst geschriebene Systeme. In verteilten Systemen sind Namenssysteme wie JNDI (Java Naming and Directory Interface) eine Möglichkeit, verteilte Komponenten innerhalb des Systems bekannt zu machen (Listing 2–5). Java EE verwendet diesen Ansatz in sehr vielen Bereichen.

```
public BestellungBusinessProcess() {  
    InitialContext initialContext =  
        new InitialContext(environment);  
    KundeDAOHome kundeDAOHome = (KundeDAOHome)  
        initialContext.lookup("kundeDAO");  
    this.kundeDAO = kundeDAOHome.create();  
    initialContext.close();  
}
```

**Listing 2-5**

Erzeugung mithilfe eines  
Namenssystems

Namenssysteme stellen einen Sonderfall des FACTORY-Patterns dar. Es wird lediglich ein Name als Parameter übergeben. Entsprechend treten auch die für den FACTORY-Ansatz typischen Probleme auf. Außerdem werden bei einer direkten Verwendung der JNDI-API große Teile des Systems von JNDI abhängig, was das Unterschieben von Mocks und die Änderbarkeit weiter erschwert. Außerdem muss man auch die Fehler aus diesen APIs behandeln, was in Listing 2-5 der Einfachheit halber nicht gemacht wurde und den Code noch komplexer machen würde, zumal es nicht einfach ist, auf solche Fehler sinnvoll zu reagieren.

Es ist auch grundsätzlich bedenklich, wenn Teile der Geschäftslogik von einer technischen API wie JNDI abhängen. Dies widerspricht nämlich der »Separation of Concerns« (Trennung der Belange), einem grundlegenden Prinzip der Softwareentwicklung. Technische Belange sollten von Geschäftslogik separiert werden, damit die Geschäftslogik auch auf andere Technologien portiert werden kann, und umgekehrt sollten die grundlegenden Technologien auch für andere Geschäftslogiken funktionieren. Nur so kann man ein System z.B. später auf neuere Technologien portieren.

Viele der gezeigten Ansätze führen zu Schwierigkeiten, weil sie zahlreiche, zum Teil auch noch versteckte Abhängigkeiten einführen und damit den Aufbau einer Testumgebung erschweren. Die Testbarkeit ist aber nur ein Aspekt des eigentlichen Problems: Man handelt sich eine verringerte Flexibilität ein, weil man die Objektnetze nur schwer umkonfigurieren kann. Außerdem werden in einigen Fällen Abhängigkeiten eingeführt, die grundsätzlich bedenklich sind, wie z.B. die Abhängigkeit zu einer technischen API wie JNDI in fachlichen Klassen.

*Keine Lösung...*

Da alle Ansätze Schwächen, aber auch Stärken haben, ist es in der Praxis meistens so, dass mehrere Ansätze parallel verwendet werden. Dadurch ist das System unnötig komplex. Wenn man an einer Stelle eine flexiblere Lösung, z.B. für die Erleichterung der Tests, gefunden hat, muss man an den anderen Stellen des Systems auch aktiv werden, falls sie eine andere Technik verwenden.

*Viel hilft viel*

Dieser Zustand ist eigentlich überraschend, da Referenzen und Delegation diejenigen Elemente der Objektorientierung sind, die am meisten verwendet werden. Sie sind deutlich wichtiger als beispiels-

*Referenzen und  
Delegation bisher  
kaum unterstützt*

weise Vererbung. Während Vererbung jedoch von den objektorientierten Sprachen direkt unterstützt wird, gibt es bei Referenzen und Delegation wie gezeigt zahlreiche Lösungen, die alle mehr oder weniger auf eine manuelle Implementierung hinauslaufen.

## 2.4 Der neue Ansatz: Dependency Injection

Eine neue Lösung zum Aufbau von Objektnetzen ist Dependency Injection (DI). Wie der Name erkennen lässt, werden die abhängigen Objekte zur Laufzeit »injiziert«, also den Objekten zugewiesen. Die Objekte sind damit zwar immer noch von anderen abhängig, aber die Abhängigkeiten sind explizit an der Schnittstelle durch die entsprechenden set-Methoden zu erkennen. Außerdem ist das Objekt passiv: Ihm werden die abhängigen Objekte zugewiesen, statt sie z.B. bei einer FACTORY aktiv zu erzeugen. Dadurch kann man dem Objekt beliebige andere Objekte zuweisen, z.B. auch Mocks oder für andere Umgebungen angepasste Objekte.

Besonders flexibel wird der Ansatz, wenn die Objekte nur noch von Interfaces abhängen. Dann kann man beliebige Objekte injizieren, die das Interface implementieren. Aber auch wenn die Klasse der Parameter festgelegt ist, gewinnt man Flexibilität. Man kann beispielsweise Instanzen einer Unterklasse der eigentlich erwarteten Klasse übergeben oder Instanzen, die mit einem anderen Zustand initialisiert wurden.

### **Tipp**

Das Programmieren gegen Interfaces statt direkte Implementierungen ist sowieso eine gute Idee. Dadurch hängen die Objekte nur noch von den Interfaces ab und nicht mehr von den Implementierungsklassen, so dass man leichter die konkrete Implementierung austauschen kann. Nachteil ist natürlich, dass man das zusätzliche Interface auch implementieren muss.

### *DI-Konfiguration*

Irgendwo muss definiert werden, was den Objekten zugewiesen werden soll. Das kann man z.B. in Java implementieren. Dann ist man aber wieder bei dem in Listing 2–1 dargestellten Zustand, der zu unübersichtlichem Code führt. Das Entscheidende ist, dass es bei den DI-Infrastrukturen üblicherweise eine externe Konfiguration gibt, in der die Beziehungen zwischen den Objekten konfiguriert werden, und die Erzeugung selbst wird nicht ausprogrammiert. Die DI-Infrastruktur erzeugt entsprechend der Konfiguration die Objekte und stellt die Verbindungen zwischen den Objekten her. Dadurch sind Probleme wie die Erzeugung in der richtigen Reihenfolge gelöst, und die Konfiguration ist auch einfacher und übersichtlicher als der Code für die Erzeugung der Objekte.

Man kann also beispielsweise in der Konfiguration definieren, dass eine Instanz von `KundeDAO` erzeugt werden soll. Außerdem soll eine Instanz von `BestellungBusinessProcess` erzeugt werden, und der Methode `setKundeDAO()` soll die bereits erzeugte Instanz von `KundeDAO` übergeben werden. In diesem einfachen Fall wäre der entsprechende Java-Code noch eine gangbare Alternative zur Konfiguration, aber die Konfiguration erlaubt eine größere Flexibilität ohne direkte Änderungen im Code, und man muss eben beispielsweise die Reihenfolge nicht beachten. Es ist aber keine Konfiguration, mit der ein Administrator die Anwendung anpassen würde, da dieser nicht an den Objekten und dem Zusammenspiel interessiert ist, sondern nur an bestimmten technischen Parametern.

Noch ein Wort zu dem Begriff »Dependency Injection«: Das Verfahren ist auch unter dem Namen »Inversion of Control« (IoC) bekannt. Allerdings ist dieser Begriff irreführend, denn es geht nicht um eine Umkehr des Kontrollflusses in der Anwendung [Fow04]. Die IoC-Eigenschaft hat jedes Framework: Es bildet einen Rahmen, der den Kontrollfluss bestimmt und selbst geschriebenen Code an wohl definierten Stellen aufruft. Hier passt der Begriff IoC, weil der eigene Code den Kontrollfluss nicht mehr festlegt, sondern nur noch passiv aufgerufen wird. Das wirklich Neue z.B. bei Spring ist durch »Dependency Injection« besser ausgedrückt: Objekte bekommen ihre abhängigen Objekte vom Spring-Framework zugewiesen, statt sie sich selbst zu suchen.

*Dependency Injection vs.  
Inversion of Control*

## 2.5 Dependency Injection mit Spring

### 2.5.1 Die Konfigurationsdatei

In der Spring-Konfigurationsdatei wird das Objektnetz mit XML konfiguriert. Die dort definierten Objekte sind ganz normale Java-Objekte. Man nennt sie auch Spring-Beans, da sie durch Spring erzeugt werden. Die grundlegenden Konfigurationsmöglichkeiten sind:

- Spring-Beans definieren: Man kann Spring-Beans durch das `bean`-Element anlegen. Sie bekommen einen Namen, und man muss definieren, von welcher Klasse die Spring-Beans sein sollen.
- Mit dem `property`-Element kann man einer Property einer Spring-Bean einen Wert zuweisen. Dahinter verbirgt sich der Aufruf einer entsprechenden `set`-Methode. Damit werden die einzelnen Spring-Beans konfiguriert. Ein Beispiel ist das Setzen des Benutzernamens für eine Datenbankverbindung. Der Wert für die Property wird in

einem value-Attribut angeben. Alternativ kann man Referenzen zwischen Objekten herstellen: Dadurch wird das Objektnetz aufgebaut. Hier wird mit dem ref-Attribut gearbeitet.

Eine Konfiguration nach diesem Schema findet sich bei der Beispielanwendung in der Datei `spring-jdbc-beans.xml` (Listing 2–6). Wie man sieht, gibt es auch eine passende DTD für die Spring-Konfiguration. Im Beispiel wird die DTD verwendet, die seit Spring 2.0 aktuell ist. Davor trugen die DTDs keine Versionsnummer.

In den Listings werden Package-Namen oft durch ... ersetzt, um die Lesbarkeit und die Übersichtlichkeit zu verbessern. Da moderne Entwicklungsumgebungen meistens das Importieren der Klassen aus den entsprechenden Pakkages automatisieren, muss man den Package-Namen oft auch gar nicht kennen.

### Listing 2–6

Die Spring-Konfigurationsdatei `spring-jdbc-beans.xml` für die Beispielanwendung

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
"http://www.springframework.org/dtd/spring-beans-2.0.dtd">

<beans>

  <bean name="datasource"
    class="org.apache.commons.dbcp.BasicDataSource"
    destroy-method="close">
    <property name="driverClassName"
      value="org.hsqldb.jdbcDriver" />
    <property name="url"
      value="jdbc:hsqldb:file:springbuchhsqldb" />
    <property name="username"
      value="sa" />
    <property name="password"
      value="" />
  </bean>

  <bean name="kundeDAO" class="springjdbcdao.KundeDAO">
    <property name="datasource" ref="datasource"/>
  </bean>

  <bean name="bestellungDAO"
    class="springjdbcdao.BestellungDAO">
    <property name="datasource" ref="datasource"/>
    <property name="kundeDAO" ref="kundeDAO"/>
    <property name="wareDAO" ref="wareDAO"/>
  </bean>

  <bean name="wareDAO" class="springjdbcdao.WareDAO">
    <property name="datasource" ref="datasource"/>
  </bean>
```

```
<bean name="bestellung"
  class="...BestellungBusinessProcess">
  <property name="bestellungDAO"
    ref="bestellungDAO"/>
  <property name="kundeDAO">
    ref="kundeDAO"/>
  <property name="wareDAO"
    ref="wareDAO"/>
</bean>

</beans>
```

Bei der Konfiguration der DataSource sieht man, wie man einem Objekt mit dem `value`-Attribut direkt Konfigurationswerte zuweisen kann. Den DAOs wird mit dem `ref`-Attribut jeweils eine Referenz auf die DataSource zugewiesen, die zum Zugriff auf die Datenbank notwendig ist. Zusätzlich werden die Beziehungen zwischen den DAOs konfiguriert, und schließlich bekommt der Geschäftsprozess alle DAOs zugewiesen. Damit ist also definiert, wie man Objektnetze und Objektkonfigurationen mithilfe von Spring festlegen kann.

Übrigens hat Spring bis einschließlich Version 1.1.x statt der `value`- und `ref`-Attribute nur das `value`- bzw. `ref`-Element bei den `property`-Elementen unterstützt:

```
<property name="driverClassName">
  <value>org.hsqldb.jdbcDriver</value>
</property>
```

bzw.

```
<property name="datasource">
  <ref bean="datasource"/>
</property>
```

Diese Schreibweise führt zu wesentlich längeren Konfigurationsdateien, so dass man sie nicht mehr verwenden sollte. Allerdings kann man beim `value`-Element mithilfe des XML-Konstrukts `CDATA` auch Zeichen verwenden, die nicht XML-konform sind. Ebenfalls kann man mit dem `type`-Attribut angeben, von welchem Typ der Wert sein soll, falls man einen bestimmten Typ erzeugen will.

Neben der DTD gibt es auch ein XML-Schema [XMLSchema] für Spring-Konfigurationen. XML-Schemas bieten im Vergleich zu DTDs wesentlich bessere Möglichkeiten für die Definition eigener Typsysteme. Das macht für den Spring-Nutzer zunächst wenig Unterschied, denn es ändern sich lediglich einige XML-Deklarationen (Listing 2–7). Aufbauend auf XML-Schemas gibt es es aber für einige Spring-Features vereinfachte Konfigurationen mit eigenen Schemas, wie in den

*XML-Schema*

folgenden Kapiteln noch erläutert werden wird. Intern werden für die DTD-basierte und die XML-Schema-basierte Konfiguration dieselben Datenstrukturen verwendet, so dass die beiden Möglichkeiten austauschbar sind und miteinander integriert werden können.

Mithilfe der XML-Schemas wird eine Möglichkeit geschaffen, die Spring-Konfiguration mit eigenen Ausdrucksmöglichkeiten zu erweitern. Das ist jedoch typischerweise vor allem für technische Konfigurationen sinnvoll. Es ist aber auch denkbar, für bestimmte fachliche Einsatzkontexte eigene Konfigurationsmöglichkeiten zu definieren. Dazu muss man einen NamespaceHandler implementieren.

### Listing 2-7

Verwendung von  
XML-Schemas statt DTDs

```
<?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">
</beans>
```

### Tip

In dem hier gezeigten Beispiel werden die Spring-Beans durch das name-Attribut mit einem Namen versehen. Stattdessen kann man auch das id-Attribut verwenden, das allerdings nur einen eingeschränkten Zeichensatz erlaubt, und man kann nur einen Namen angeben. Dieses Attribut ist als XML-ID-Typ definiert, der für die eindeutige Identifikation von XML-Elementen vorgesehen ist.

Es ist auch möglich, dass man weder id noch name deklariert. Dann wird als Name der Bean automatisch der Klassenname und eine laufende Nummer verwendet. So ergeben sich natürlich keine sprechenden Namen.

Man kann später noch Beans mit einem Alias-Namen versehen, also einen zusätzlichen Namen angeben:

```
<bean id="originalName"></bean>
<alias name="originalName" alias="aliasName" />
```

Dadurch kann man Spring-Beans zusätzliche sprechende Namen geben oder die Konfiguration flexibel halten, indem man mehrere Referenzen nicht direkt auf eine Bean zeigen lässt, sondern auf einen Alias, den man leicht umkonfigurieren kann. Außerdem kann man so andere Bean-Konfigurationen integrieren, bei denen die Bean-Namen nicht wie erwartet gewählt wurden.

Will man den Namen einer Bean in der Spring-Konfiguration angeben, kann man natürlich einfach nur den Namen als String in der Konfiguration angeben. Dann läuft man allerdings Gefahr, dass man sich vertippt und dann später feststellt, dass keine Spring-Bean mit dem Namen existiert. Wenn man das idref-Element mit dem local-Attribut und bei der Spring-Bean die id definiert, findet bereits der XML-Parser solche Fehlkonfigurationen. Gute XML-Editoren können solche Fehler auch darstellen. Hier ein Beispiel:

```

<bean id="datasource" ...>
...
</bean>

<bean id="...">
  <property name="BeanNameDerDatasource">
    <idref local="datasource"/>
  </property>
</bean>

```

Will man die Namen der Spring-Beans mit dem name-Attribut referenzieren, muss man beachten, dass man statt

```
<idref local="datasource" />
```

Folgendes verwenden muss:

```
<idref bean="datasource" />
```

Eine weitere Möglichkeit für die Konfiguration ist die Verwendung des p-Namespace. An dem Namen kann man schon erkennen, worum es geht: Kürze. Im Gegensatz zu den anderen Konfigurationsmöglichkeiten wird in diesem Fall jede Property nur durch ein Attribut des bean-Elements definiert und nicht durch ein eigenes Element. Ein Beispiel zeigt Listing 2–8. Wie man sieht, kann man durch das p-Präfix mit einem XML-Attribut die Property einer Spring-Bean setzen. Wenn man noch ein -ref anhängt, kann man so auch Referenzen setzen. Dadurch wird die Konfiguration noch kompakter. Ob man eine Konfiguration mit dem p-Namespace auch übersichtlich findet, muss man ausprobieren: Die sonst verwendeten property-Elemente können eine bessere Lesbarkeit bieten.

*Einfachere Konfiguration  
mit dem p-Namespace*

```

<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="datasource"
    class="...BasicDataSource"
    destroy-method="close"
    p:driverClassName="org.hsqldb.jdbcDriver"
    p:url="jdbc:hsqldb:file:springbuchhsqldb"
    p:username="sa"
    p:password="" />

```

**Listing 2–8**  
*Konfiguration mit  
dem p-Namespace*

```

<bean id="kundeDAO"
      class="...KundeDAO"
      p:dataSource-ref="datasource" />
</beans>

```

### 2.5.2 Die BeanFactory

Um im Code auf die konfigurierten Spring-Beans zugreifen zu können, bietet Spring die BeanFactory an. Dieser muss die Konfigurationsdatei übergeben werden, und anschließend kann man sich von der BeanFactory eine Referenz auf Spring-Beans zurückgeben lassen. Listing 2–9 zeigt den dazu notwendigen Code.

#### Listing 2–9

Auslesen eines Objekts aus der Spring-Konfiguration

```

ClassPathResource res =
    new ClassPathResource("beans.xml");
BeanFactory beanFactory =
    new XmlBeanFactory(res);
BestellungBusinessProcess bestellung;
bestellung = beanFactory.getBean("bestellung",
    BestellungBusinessProcess.class);

```

Die Klasse `ClassPathResource` ist eine Spring-eigene Implementierung einer Ressource, die aus dem Classpath ausgelesen wird. Alternativ könnte man hier auch eine `FileSystemResource` verwenden, bei der die Spring-Konfiguration aus dem Dateisystem ausgelesen wird (Abschnitt 2.9.1).

Hinter den Kulissen wird die Konfiguration eingelesen, und es werden die notwendigen Objekte erzeugt, in diesem Fall also der Geschäftsprozess selbst, die abhängigen DAOs und schließlich die `DataSource`. Würde jetzt ein zweites Mal ein `BestellungBusinessProcess` ausgelesen, so wird dasselbe Objekt zurückgegeben. Die Spring-Beans sind also SINGLETONS.

#### Tipp

Eigentlich widerspricht das Auslesen eines Objekts dem Prinzip der Dependency Injection, da hier eine FACTORY verwendet wird. Daher sollte es möglichst sparsam verwendet werden, also nur in der `main()`-Methode. Außerdem sollte man möglichst wenige Objekte auslesen. Im Beispiel muss man auf jeden Fall den Geschäftsprozess auslesen, um ihn zu nutzen. Die DAOs sind z.B. bei Testfällen für den Aufbau der Testdaten ebenfalls unumgänglich. Ein direkter Zugriff auf die `DataSource` hingegen ist kaum zu rechtfertigen. Abschnitt 2.7 beschäftigt sich noch detailliert damit, wann man Dependency Injection verwenden kann, denn man kann das Auslesen der Objekte weitestgehend vermeiden – in der Praxis ist es nur in wenigen Ausnahmefällen notwendig.

### 2.5.3 Constructor Dependency Injection

In der Datei wurde die Konfiguration dadurch vorgenommen, dass die abhängigen Objekte als Properties gesetzt werden, also durch den Aufruf von set-Methoden (Setter Dependency Injection). Spring bietet als Alternative die Möglichkeit, die Referenzen als Parameter an den Konstruktor zu übergeben. Dies nennt man Constructor Dependency Injection. Dann muss man die Konfigurationsdatei anpassen. Listing 2–10 zeigt die Konfiguration einer Spring-Bean aus der Beispielanwendung mit Constructor Dependency Injection.

```
<bean id="bestellung"  
  class="...BestellungBusinessProcess">  
  
  <constructor-arg ref="bestellungDAO"/>  
  <constructor-arg ref="kundeDAO"/>  
  <constructor-arg ref="wareDAO"/>  
</bean>
```

Die Reihenfolge, in der die Konstruktorparameter angegeben werden, spielt keine Rolle. Die Parameter werden anhand des Typs erkannt und in die richtige Reihenfolge gebracht. Das kann natürlich nur funktionieren, wenn die Typen der Konstruktorparameter sich auch unterscheiden. Sonst kann man die Reihenfolge explizit mit dem `index`-Attribut angeben. Bei dem ersten Parameter wäre dies der Index 0:

```
<constructor-arg index="0" ref="bestellungDAO"/>
```

Ein weiteres Problem tritt auf, wenn man primitive Datentypen verwendet. Dann ist es nämlich auch nicht eindeutig, von welchem Typ der Ausdruck überhaupt ist. In diesem Fall kann man den Typ explizit angeben. Dazu verwendet man das `type`-Attribut:

```
<constructor-arg index="0"  
  type="int"  
  value="42" />  
<constructor-arg index="1"  
  type="java.lang.String"  
  value="42" />
```

Mit dem `name`-Attribut kann man einen Konstruktorparameter auch dem Namen nach auswählen. Das funktioniert allerdings nur, wenn man die Anwendung mit Debug-Informationen übersetzt hat, da sonst die Namen der Konstruktorparameter zur Laufzeit nicht mehr zur Verfügung stehen:

```
<constructor-arg name="ganzzahl"  
  value="42" />  
<constructor-arg name="zeichenkette"  
  value="42" />
```

#### Listing 2–10

*Bean-Konfiguration mit Constructor Dependency Injection: ein Ausschnitt der `jdbc-beans-constructor.xml`-Datei aus dem Beispiel*

**Tipp**

Konstruktor-DI ist sinnvoll, wenn die Spring-Beans ohne abhängige Objekte nicht benutzbar sind. Der Vorteil von Constructor Dependency Injection ist, dass man immer nur Spring-Beans mit allen notwendigen abhängigen Objekten erzeugen kann und dadurch Fehlkonfigurationen ausschließt. In der Beispielanwendung gilt dies unter anderem für die DAOs: Ohne Referenz auf die DataSource kann man sie nicht verwenden. Optionale Konfigurationen oder Referenzen können hingegen durch Setter-DI eingefügt werden. Das funktioniert auch als Ergänzung, wenn man Konstruktor-DI für die notwendigen Referenzen und Konfigurationen verwendet.

Setter-DI hat auch Vorteile: Durch die JavaDoc-Kommentare der set-Methoden sind die einzelnen Properties leicht zu dokumentieren. Außerdem kann man Vorgabewerte definieren, die mit einer set-Methode überschrieben werden können. Die Properties werden auch an Subklassen vererbt, während man bei Konstruktor-DI den Konstruktor der Oberklasse explizit aufrufen muss.

**2.5.4 Erzeugung mit Factories**

Gerade bei der Umstellung von vorhandenem Code auf Spring kommt es vor, dass der alte Code FACTORIES verwendet. In diesen Fällen funktionieren die bisher vorgestellten Vorgehensweisen zur Erzeugung der Spring-Beans mithilfe eines Konstruktors nicht. Spring bietet für vorhandene FACTORIES eine Lösung, da man die Erzeugung von Spring-Beans mit Factories konfigurieren kann.

*Factories mit statischen  
oder Instanzmethoden*

Nehmen wir z.B. an, dass in der Beispielanwendung bisher die Erzeugung der DAOs durch statische Methoden in der Klasse StaticDAOFactory gelöst worden ist und die Anwendung nun auf Spring umgestellt werden soll. In diesem Fall könnte folgende Konfiguration für ein DAO erfolgen:

```
<bean id="kundeDAO"
      class="jdbcdao.StaticDAOFactory"
      factory-method="createKundeDAO"/>
```

Die Spring-Bean wird nun also durch die Methode createKundeDAO() der Klasse StaticDAOFactory erzeugt. Es ist ebenfalls möglich, statt einer Klasse ein Objekt als FACTORY zu verwenden. Wenn also die Erzeugung nicht durch eine statische Methode, sondern durch eine Instanzmethode stattfindet, würde man folgende Konfiguration verwenden:

```
<bean id="DAOFactory"
      class="jdbcdao.InstanceDAOFactory"/>
<bean id="kundeDAO"
      factory-bean="DAOFactory"
      factory-method="createKundeDAO"/>
```

Es wird zuerst eine Instanz von `InstanceDAOFactory` erzeugt und an dieser wird `createKundeDAO()` aufgerufen. Das Ergebnis ist dann der Spring-Bean `kundeDAO`. Man kann mit dem `constructor-arg`-Element auch Parameter für die Factory-Methode definieren.

Diese Art der Integration von Factories hat allerdings einige Nachteile. So sind Typinformationen über die Spring-Beans nur zu bekommen, indem man eine Spring-Bean mithilfe der `FACTORY` erzeugt und anschließend den Typ überprüft. Daher werden Autowiring-Methoden (Abschnitt 2.8), bei denen aufgrund von Typinformationen die Beziehungen zwischen den Spring-Beans automatisch hergestellt werden, mit solchen `FACTORIES` nicht unterstützt. Auch verliert man Informationen darüber, ob die `FACTORY` immer dasselbe Objekt zurückgibt, also ein `SINGLETON` implementiert. Dadurch kann Spring solche Objekte nicht wie andere `SINGLETONS` behandeln. Sie werden also z.B. nicht beim Aufruf von `preInstantiateSingletons()` an der `BeanFactory` instanziiert.

Spring definiert daher ein Interface, das `FACTORIES` implementieren sollten, die Spring selbst verwaltet (Listing 2–11). Man nennt solche Objekte `FactoryBeans`. Neben der eigentlichen Erzeugung durch die `getObject()`-Methode liefern `FactoryBeans` auch Informationen über die erzeugten Objekte. Die `getObjectType()`-Methode liefert den Typ der erzeugten Objekte und `isSingleton()` die Information, ob immer dasselbe Objekt zurückgegeben wird. Dadurch kann Spring beispielsweise Autowiring implementieren. `FactoryBeans` sind ein recht mächtiges Werkzeug, um Schnittstellen zu anderen `FACTORY`-Mechanismen wie beispielsweise `JNDI` zu implementieren.

```
public interface FactoryBean<T> {
    T getObject() throws Exception;
    Class<? extends T> getObjectType();
    boolean isSingleton();
}
```

#### Listing 2–11

Interface für Spring  
`FactoryBeans`

In der Konfiguration wird die `FactoryBean` wie sonst die Klasse einer Spring-Bean angegeben. Spring stellt dann fest, dass diese Klasse eine `FactoryBean` ist und erzeugt zunächst eine Instanz der `FactoryBean`. Sie wird anschließend verwendet, um die eigentliche Spring-Bean zu erzeugen. Wenn man eine Referenz der `FactoryBean` einem anderen Objekt per DI zuweist, wird hinter den Kulissen also eine neue Spring-Bean erzeugt und der anderen Spring-Bean zugewiesen. Die folgende Konfiguration zeigt, wie das DAO für die Kunden von einer `FactoryBean` `KundeDAOFactoryBean` erzeugt werden könnte:

```
<bean id="kundeDAO"
      class="KundeDAOFactoryBean"/>
```

Es wird in diesem Fall eine Instanz der Klasse `KundeDAOFactoryBean` erzeugt. An dieser Instanz wird `getObject()` aufgerufen. Anschließend wird es entsprechend den DI-Regeln weiterverarbeitet, also z.B. per `set`-Methode einer Property zugewiesen.

Ein Zugriff auf die `FactoryBean` selbst ist also nicht ohne Weiteres möglich. Man bekommt ja jedes Mal nur ein Produkt geliefert, nicht die `FactoryBean` selbst. Um dennoch eine Referenz auf die `FactoryBean` zu bekommen, kann man der `BeanFactory` beim Aufruf von `getBean()` beim Namen noch ein kaufmännisches Und (`»&«`) vor dem Namen der `FactoryBean` übergeben. In diesem Fall bekommt man tatsächlich eine Referenz auf die `FactoryBean` und nicht auf ein Produkt:

```
BeanFactory beanFactory = ...
FactoryBean<KundeDAO> factoryBean =
    (FactoryBean<KundeDAO>)
    beanFactory.getBean("&kundeDAO");
```

**Tip**

Die Erzeugung mithilfe von statischen Methoden oder Instanzmethoden ist vor allem für die Integration von älterem Code oder Bibliotheken sinnvoll. Man kann allerdings auch jeweils eine passende `FactoryBean`-Implementierung schreiben. Wenn man in der Verlegenheit ist, eine `FACTORY` in einer Spring-Anwendung selbst zu entwickeln, sollte man auf jeden Fall das `FactoryBean`-Interface implementieren, da die Verwendung in der Konfiguration einfacher ist und Spring-Features wie Autowiring (Abschnitt 2.8) nur so nutzbar sind. Eine `FactoryBean` kann auch je nach Konfiguration unterschiedliche Produkte erzeugen. Daher werden `FactoryBeans` für den konfigurierbaren Zugriff auf generische `FACTORY`-Mechanismen wie JNDI verwendet. Eine einzige `FactoryBean`-Klasse ist dann als Schnittstelle ausreichend. Für JNDI bietet Spring passende Klassen wie z.B. die `JndiObjectFactoryBean` an. Der `FactoryBean`-Mechanismus ist also eine mächtige Möglichkeit, andere Technologien in Spring zu integrieren.

In Spring sind einige interessante `FactoryBeans` integriert:

- Die `MethodInvokingFactoryBean` bietet eine Alternative zum `factory-method`-Attribut im `bean`-Element. In der Property `targetMethod` wird die aufzurufende Methode festgelegt und in der Property `argument` die Parameter. Die Property `targetObject` definiert die aufzurufende Spring-Bean. Statische Methodenaufrufe sind auch möglich, dazu dient die Property `targetClass`. Das Ergebnis dieses Aufrufs darf im Gegensatz zum `factory-method`-Attribut des `bean`-Elements in der Spring-Konfiguration auch `null` oder `void` sein. Man kann also mit diesem Mechanismus Methodenaufrufe in einer Spring-Konfiguration festlegen.
- Die `ServiceLocatorFactoryBean` bietet die Möglichkeit, ein vorhandenes Interface zum Erzeugen von Objekten an Spring »umzulen-

ken«, also die Aufrufe an die BeanFactory zu delegieren. Dazu muss man das Interface der Property serviceLocatorInterface zuweisen. Die ServiceLocatorFactoryBean erzeugt dann die Implementierung dieses Interfaces. Wenn nun ein Aufruf an eine Methode aus diesem Interface geschieht, wird bei einem Aufruf ohne Parameter oder mit null bzw. leerem String einfach ein typkompatibles Spring-Bean aus der Konfiguration herausgesucht und zurückgegeben. Sonst wird der Parameter in einen String umgewandelt, und es wird eine Spring-Bean mit diesem Namen aus der Spring-Konfiguration ausgelesen. Mit dieser Klasse ist es also möglich, eine vorhandene Anwendung mit einer FACTORY auf Spring umzustellen: Man verwendet statt der alten Factory eine ServiceLocatorFactoryBean, die das Interface der bisherigen FACTORY implementiert und die Aufrufe an die BeanFactory delegiert. Die Anwendung selbst merkt dann nicht, dass statt der alten FACTORY in Wirklichkeit eine BeanFactory verwendet wird.

## 2.6 Vorteile von DI

### 2.6.1 Wir rufen Sie an...

Durch die Konfiguration mithilfe der Spring-BeanFactory erreicht man, dass der Code einer Klasse nicht mehr direkt von der Umgebung abhängt, da alle Beziehungen zur Umgebung durch die BeanFactory hergestellt werden.

In der Beispielanwendung betrifft das sowohl die Beziehungen der Objekte, die Services anbieten, wie z.B. die DAOs und der Geschäftsprozess, als auch die Beziehungen zu technischen Objekten wie z.B. der DataSource. DI ist die Umkehrung des sonst üblichen Prinzips, bei dem jedes Objekt die Beziehungen zu anderen Objekten selbst aufbaut. Eine Metapher für DI ist das Hollywood-Prinzip: Am Ende des Vorstellungsgesprächs eines Schauspielers für einen Film steht meistens die Aussage: »Rufen Sie uns nicht an, wir rufen Sie an.« Genauso ist es hier mit den Objekten: Sie rufen nicht aktiv die Umgebung auf, sondern werden von ihr mit den notwendigen Referenzen versehen. Wobei hier – im Gegensatz zu den Hollywood-Gesprächen – der Anruf tatsächlich erfolgt.

Das ergibt den Vorteil, dass die Objekte von zwei Aufgaben befreit sind, nämlich der Konfiguration und dem Aufbau der Umgebung. Dadurch wird der Code völlig unabhängig von der Umgebung. Das entspricht dem Ziel von Spring, möglichst wenig invasiv zu sein, also den Code möglichst wenig zu beeinflussen. Aus Dependency Injection ergeben sich aber noch weitere Vorteile.

### 2.6.2 Flexibilität

Ein Vorteil dieses Vorgehens ist die erhöhte Flexibilität. So wird durch die externe Konfiguration der Austausch einer Implementierung trivial. In der Beispielanwendung kann dies an verschiedenen Stellen von Vorteil sein:

- Die Persistenz ist in den DAOs vollständig gekapselt. Wenn man eine neue Implementierung der DAOs mit einer anderen Technologie entwickelt, muss man für den Einsatz dieser neuen Persistenzschicht lediglich die Konfiguration so modifizieren, dass die neuen DAOs verwendet werden.
- Der Zugriff auf die DataSource erfolgt im Moment durch die Erzeugung einer eigenen DataSource (Listing 2–6). In einem Java-EE-Kontext hinterlegt der Application-Server die DataSource im JNDI-Namenssystem, und man muss sie dort unter einem vorgegebenen Namen auslesen. Will man eine Spring-Anwendung auf einem Application-Server laufen lassen, muss man lediglich in der Konfiguration die passenden Änderungen vornehmen. Dazu kann eine FactoryBean (Abschnitt 2.5.4) verwendet werden, die das Objekt aus dem JNDI-Kontext ausliest (Listing 2–12). Dadurch kann also die Anwendung ohne Änderung des Quellcodes in einer anderen technischen Umgebung ablauffähig gemacht werden.

#### Listing 2–12

Auslesen eines Objekts aus dem JNDI-Namenssystem mit Spring

```
<bean id="datasource"
      class="...JndiObjectFactoryBean">
  <property name="jndiName"
            value="java:comp/env/jdbc/DB" />
</bean>
```

- Da die Klassen ihre Abhängigkeiten zugewiesen bekommen, ist es ohne Weiteres möglich, die Klassen sogar völlig ohne eigene Infrastruktur – also auch ohne Spring-Framework – zu testen. Dazu kann man einen Testrahmen entwickeln, der z.B. mit EasyMock [EasyMock] aufgesetzt wird. Dies wird in Abschnitt 2.15.1 näher erläutert.

### 2.6.3 Eingebaute Konfigurierbarkeit

Die Spring-Konfigurationsdatei ist recht feingranular: Sie definiert die Beziehungen zwischen Objekten, also den kleinsten Einheiten eines objektorientierten Systems. Für einen Entwickler ist das ideal, aber es ist nicht das, was beispielsweise ein Administrator für Änderungen an einer Anwendung verwenden will. Wie Abschnitt 2.10.6 zeigt, gibt es aber Möglichkeiten, Konfigurationen aufzuteilen oder mithilfe von

Property-Dateien nur jene Einstellungen konfigurierbar zu halten, die ein Administrator bei seiner Arbeit auch ändern muss (Listing 2–29 und 2–31).

Auf jeden Fall ist durch Spring ein Konfigurationsmechanismus vorhanden, den man nicht selbst entwickeln muss. Die Spring-Beans müssen sich nur die Einstellungen zuweisen lassen, um konfigurierbar zu sein. Außerdem ist der Konfigurationsmechanismus konsistent für die gesamte Anwendung. Die Konfiguration einer FACTORY mit einer Konfigurationsdatei ist auch in Projekten üblich, die Spring nicht nutzen, um die Produkte der FACTORY leichter ändern zu können. Wenn jedoch mehrere FACTORIES im System vorhanden sind, führt dies häufig zu einer großen Anzahl von Konfigurationsdateien, die einzeln verwaltet werden müssen und oft auch eine unterschiedliche Syntax haben.

#### 2.6.4 Das Singleton-Pattern

In Spring sind alle Spring-Beans zunächst SINGLETONS: Es gibt nur eine Instanz. Das ist vielleicht überraschend, denn in den üblichen objektorientierten Programmiersprachen ist das Gegenteil der Fall: Man kann beliebig viele Instanzen einer Klasse erzeugen. Man muss schon das SINGLETON-Pattern anwenden, um dies zu ändern. Aber wenn man sich die Beispielanwendung anschaut, sieht man, dass die Objekte dort zum größten Teil nur Dienste anbieten und daher eine Instanz vollkommen ausreichend ist. Sie haben typischerweise keinen Zustand, der spezifisch für einen Client ist, sondern alle notwendigen Informationen werden als Parameter übergeben. Hätten sie einen eigenen Zustand, könnten sie nur als SINGLETON implementiert werden, wenn sie Thread-sicher sind. Dies ist der Fall, wenn parallele Zugriffe nicht zu irgendwelchen Konflikten führen können. Solche parallelen Zugriffe sind natürlich bei Serveranwendungen eher die Regel als die Ausnahme. Dennoch sind die meisten Objekte Thread-sicher, zumindest wenn sie wie im Beispiel Dienste so anbieten, dass sie alle Informationen als Parameter übergeben bekommen und keinen eigenen Zustand haben.

Kann tatsächlich eine Spring-Bean nicht als SINGLETON verwendet werden, gibt es einige Möglichkeiten, mit diesem Problem umzugehen (Abschnitt 3.6.3). Man kann außerdem die Spring-Beans für Webanwendungen z.B. in der HTTP-Session oder dem HTTP-Request unterbringen (Abschnitt 7.13).

Eine weitere Möglichkeit ist, die Spring-Bean als Prototype zu definieren. Das bedeutet, dass bei jedem Aufruf von `getBean()` auf der `BeanFactory` bzw. für jede Verwendung der Spring-Bean in der Spring-

*Prototypes statt  
Singletons*

Konfiguration eine neue Instanz erzeugt wird. Das ist allerdings nur in einigen Ausnahmefällen sinnvoll, z.B. wenn man für jede Abarbeitung eines Requests eine neue Spring-Bean erzeugen will oder wenn man Geschäftsobjekte durch Spring erzeugen lassen will. Insbesondere hilft es nicht bei Multi-Threading-Problemen. Würde man im Beispiel das `bestellungDAO` als Prototype definieren, würde sich sogar gar nichts ändern: Nur der `BestellungBusinessProcess` hat eine Referenz auf dieses DAO, so dass nach wie vor nur eine Instanz erzeugt wird. Es würden immer noch mehrere Threads diese eine Instanz verwenden, so dass dieselben Probleme entstehen würden, wenn das `bestellungDAO` nicht Thread-sicher ist.

Um eine Spring-Bean als Prototype zu definieren, muss man in das bean-Element das Attribut `scope` einführen:

```
<bean id="prototype-bean" scope="prototype" />
```

Gibt man dieses Attribut nicht an, so wird `scope="singleton"` angenommen. Auch die Definition eigener Scopes ist möglich.

*Singleton ohne  
Nebenwirkungen*

Bei Spring führt die Anwendung des SINGLETON-Patterns nur dazu, dass es genau eine Instanz des gewählten Objekts gibt. Das normale SINGLETON-Pattern verletzt Prinzipien der Objektorientierung, weil man die Instanz an einem wohlbekanntem Ort findet und dadurch überall verwenden kann. Das ist praktisch dasselbe wie eine globale Variable in imperativen Sprachen. Das SINGLETON wird typischerweise durch eine statische Methode zugreifbar. Dadurch können alle Klassen Referenzen auf das SINGLETON haben, ohne dass es unmittelbar zu erkennen ist. Bei dem Spring-Vorgehen ist die Verwendung eines normalen Objekts oder eines SINGLETONS gleich: Die Spring-Bean wird in beiden Fällen durch Dependency Injection zugewiesen. Das bedeutet, dass SINGLETONS durch eine einfache Änderung der Konfiguration zu »normalen« Objekten mit mehreren unterschiedlichen Instanzen gemacht werden können.

*Instanzmethoden statt  
statischer Methoden*

Außerdem wird es einem Entwickler durch den einfachen Umgang mit SINGLETONS erleichtert, Logik in Instanzmethoden zu implementieren, auch wenn die Objekte keinen eigenen Zustand haben und man daher die Logik auch in statischen Klassenmethoden hätte implementieren können. So kann man auch solche Methoden für Tests durch Mocks ersetzen oder Unterklassen bilden, in denen die Methoden überschrieben werden. Das ist bei statischen Methoden nicht möglich.

*Weniger Instanzen*

Oft werden auch unnötig viele Instanzen eines Objekts erzeugt, obwohl sie keinen eigenen Zustand haben. Auch hier profitiert man davon, dass Objekte bei Spring ohne weitere Konfiguration SINGLETONS sind. Dadurch kann man bei solchen zustandslosen Objekten mit

dieser einen Instanz auskommen. Natürlich darf dabei nicht im Mittelpunkt die Optimierung der Instanz-Anzahl stehen, sondern die Sicherheit der Anwendung, zumal man durch die geringere Anzahl an Instanzen keinen großen Performance-Gewinn erreichen wird.

Ingesamt kann man also SINGLETONS durch Spring mit weniger Problemen nutzen, da sie auf die Eigenschaft, dass sie nur eine Instanz haben, reduziert sind und keine anderen Nebeneffekte haben. Außerdem wird man durch die einfachere Implementierung des SINGLETON-Patterns dazu animiert, Logik nicht in statischen Methoden zu implementieren, sondern in einem SINGLETON. Das hat den Vorteil, dass die Methoden leicht durch Unterklassen überschrieben werden können, was das System flexibler macht und auch eher objektorientierten Vorstellungen entspricht.

### 2.6.5 Das Factory-Pattern

Bereits in Abschnitt 2.5.4 wurde gezeigt, wie man vorhandene FACTORIES in Spring-Anwendungen integrieren kann und mit FactoryBeans elegant eigene FACTORIES für Spring-Anwendungen implementieren kann.

Manchmal möchte man aber die Erzeugung eines Objekts entsprechend dem FACTORY-Pattern an Spring delegieren, statt selbst das Objekt zu erzeugen. Ein Grund kann die bessere Konfigurierbarkeit sein. Dazu könnte man die BeanFactory selbst aufrufen, z.B. indem man sich die Referenz auf die BeanFactory übergeben lässt. Dazu wird das Interface BeanFactoryAware in Abschnitt 2.10.4 eingeführt. Das Auslesen von Spring-Beans aus der BeanFactory hat jedoch einige Nachteile. So wird dadurch die Klasse von Spring abhängig, da sie ein von Spring definiertes Interface implementiert und sich darauf verlässt, Zugriff auf die Spring-BeanFactory zu bekommen. Zudem verletzt man das Dependency-Injection-Prinzip, da man sich die Spring-Bean nicht injizieren lässt, sondern selbst ausliest.

Innerhalb der Beispielanwendung ist die Erzeugung der Geschäftsobjekte, die beispielsweise einen Kunden modellieren, nicht mit Spring implementiert. Sie müssen mit `new` erzeugt werden und können z.B. der `save()`-Methode eines DAOs übergeben werden. Dieses Vorgehen ist bei Spring-Anwendungen üblich, denn es gibt meistens keinen Grund, die Verwaltung solcher Objekte dem Spring-Framework zu überlassen. Spring fokussiert auf die Verwaltung von Objekten, die Dienste erbringen, wie die DAOs oder die Geschäftsprozesse.

Sollte man allerdings entscheiden, dass auch diese fachlichen Objekte durch Spring kontrolliert werden sollen, um beispielsweise zu

*Objekterzeugung an die  
BeanFactory delegieren*

*Method Injection*

konfigurieren, welche Klasse instanziiert wird, oder um auch diese Objekte mit Aspekten aus dem Spring-AOP-Framework (Kapitel 3) versehen zu können, bietet Spring die Möglichkeit an, eine Methode innerhalb einer Klasse so zu überschreiben, dass sie entsprechende Produkte aus der Spring-Konfiguration erzeugt. Das wird auch Method Injection genannt:

```
<bean id="BestellungBusinessObject"
      class="businessobjects.Bestellung"
      scope="prototype" />

<bean id="bestellungDAO"
      class="dao.BestellungDAO" >
  <lookup-method
    name="createBestellung"
    bean="BestellungBusinessObject" />
</bean>
```

Mit dieser Konfiguration wird also im Objekt `bestellungDAO` die Methode `createBestellung()` so überschrieben, dass sie jeweils eine neue Instanz der Klasse `businessobjects.Bestellung` zurückgibt. Dazu wird eine Subklasse von `BestellungDAO` mit der passenden Methoden-Implementierung erzeugt, so dass die Original-Klasse nicht `final` sein darf. Außerdem muss man bei der Verwendung von Methoden wie `getClass()` vorsichtig sein, da sie ein anderes Ergebnis liefern.

Der Vorteil dieses Vorgehens ist, dass man zwar das Objekt durch die `BeanFactory` erstellen lässt, aber im Code keine Abhängigkeiten zur `BeanFactory` hat, sondern alles läuft im Verborgenen ab. Somit können also auch die Stellen, an denen man sonst möglicherweise das `FACTORY`-Pattern ausimplementieren würde, durch die `BeanFactory` mit abgedeckt werden.

Objekte unter die Kontrolle von Spring zu bringen ist auch anders möglich (Abschnitt 2.10.5), dann aber nicht mithilfe eines `FACTORY`-Ansatzes.

## 2.7 Die Grenzen von Dependency Injection

Wie im vorausgegangenen Abschnitt erläutert, ist es besser, ein Objektnetz mithilfe von `Dependency Injection` aufzubauen, als die Objekte selbst zu erzeugen oder aus der `BeanFactory` auszulesen. Irgendwo im System wird man dies jedoch tun müssen. Die Frage ist, wo genau man `Dependency Injection` nicht mehr verwenden kann und `Spring-Beans` aus der `BeanFactory` auslesen muss.

Es sollte bereits klar sein, dass die Schichten mit Serviceobjekten komplett durch `Dependency Injection` aufgebaut werden können. Im