Web Beans: Java Kontexte und "Dependency"-Einspeisung

Der neue Java Standard für "Dependency"-Einspeisung und kontextuelles Status-Management

Gavin King
JSR-299 Spezifikations-Lead
Red Hat Middleware LLC

Pete Muir
Web Beans (JSR-299 Referenz Implementation)-Lead
Red Hat Middleware LLC

David Allen
Italienische Übersetzung: Nicola Benaglia, Francesco Milesi
Spanische Übersetzung: Gladys Guerrero

Red Hat Middleware LLC
Koreanische Übersetzung: Eun-Ju Ki,
Red Hat Middleware LLC
Chinesische Übersetzung (traditionell): Terry Chuang
Red Hat Middleware LLC
Simplified Chinese Translation: Sean Wu
Kava Community

Hinweis

Der Name von JSR-299 wurde vor kurzem von "Web Beans" zu "Java Kontexte und Dependency-Einspeisung" geÃ¤ndert. Dieses Handbuch bezieht sich nach wie vor auf JSR-299 als "Web Beans" und die JSR-299 Referenzimplementierung als "Web Beans RI". Andere Dokumentation wie Blogs, Postings in Foren usw. verwenden mÃ¶glicherweise bereits die neue Namensgebung, darunter den neuen Namen fÃ¼r die JSR-299 Referenzimplementierung - "Web Beans".

You'll also find that some of the more recent functionality to be specified is missing (such as producer fields, realization, asynchronous events, XML mapping of EE resources).

Teil I. Verwendung kontextueller Objekte

Die Web Beans (JSR-299) Speziifikation definiert einen Satz von Diensten für die Java EE Umgebung, der die Entwicklung von Anwendungen maßgeblich vereinfacht. Web Beans schichtet ein verbessertes Lebenszyklus- und Interaktionsmodell über bestehende Java-Komponententypen, einschließlich JavaBeans und Enterprise Java Beans. Zur Vervollständigung des traditionellen Java EE Programmiermodells bieten Web Beans Dienste:

- einen verbesserten Lebenszyklus für stateful Komponenten, die an gut definierte *Kontexte* gebunden sind,
- eine typensichere Herangehensweise an *Dpendency-Einspeisung*,
- Interaktion über eine *Ereignisbenachrichtigungs*-Facility und
- eine bessere Vorgehensweise bei der Bindung von *Interzeptoren* an Komponenten sowie eine neue Art von Interzeptor namens *Dekorator*, der für die Lösung von Business Problemen geeigneter ist.

Dependency-Einspeisung sowie kontextuelles Lebenszyklus-Management erspart dem Benutzer eines unbekannten API das Stellen und die Beantwortung folgender Fragen:

- was ist der Lebenszyklus dieses Objekts?
- wieviele simultane Clients kann es besitzen?
- ist es multithreaded?
- wo kann ich eines bekommen?
- muss ich es explizit löschen?
- wo sollte ich meinen Verweis darauf aufbewahren, wenn ich es nicht direkt verwende?
- wie kann ich ein Indirection-Layer hinzufügen, damit die Implementierung dieses Objekts zum Zeitpunkt des Deployment variieren kann?
- wie kann ich dieses Objekt mit anderen Objekten teilen?

Ein Web Bean legt nur Typ und Semantik anderer Web Beans fest, von denen es abhängt. Es benötigt keine Informationen zum tatsächlichen Lebenszyklus, konkreter Implementierung, dem Threading-Modell oder anderen Clients eines Web Beans von dem es abhängt. Besser noch - die konkrete Implementierung, der Lebenszyklus und das Threading-Modell eines Web Beans von dem es abhängt können je nach Deployment-Szenario variieren, ohne dass dies Auswirkungen auf irgendeinen Client hätte.

Ereignisse, Interzeptoren und Dekoratoren verbessern die *lose Paarung*, die diesem Modell innewohnt:

- *Ereignisbenachrictigungen* entkoppeln Ereignis-Producer von Ereignis-Consumern,
- *Interzeptoren* entkoppeln technische Probleme von Business-Logik und
- *Dekoratoren* erlauben die Kompartmentalisierung von Business Problemen.

Und das Wichtigste - Web Beans bieten all diese Facilities auf *typensichere* Weise. Web Beans verwenden nie string-basierte Bezeichner, um zu bestimmen, wie zusammenarbeitende Objekte zusammenpassen. Und XML wird - obwohl es nach wie vor eine Option bleibt - wird selten verwendet. Stattdessen verwenden Web Beans die bereits im Java-Objektmodell verfügbaren Typinformationen gemeinsam mit einem neuen Muster namens *Binding-Annotationen*, um Web Beans, deren Abhängigkeiten, deren Interzeptoren und Dekoratoren sowie deren Ereignis-Consumer zu verbinden.

Die Web Beans Dienste sind allgemein und wenden folgende Komponententypen an, die in der Java EE Umgebung existieren:

- alle JavaBeans,
- alle EJBs und
- alle Servlets.

Web Beans bieten sogar die nötigen Integrationspunkte, so dass andere Arten von Komponenten, die durch zukünftige Java EE Spezifikationen oder nicht standardmäßige Frameworks definiert werden, sauber mit Web Beans integriert werden sowie die Web Beans Dienste nutzen und mit anderen Arten von Web Beans interagieren können.

Web Beans wurden durch eine Reihe bestehender Java Frameworks beeinflusst, darunter Seam, Guice und Spring. Jedoch besitzen Web Beans ihre eigenen Eigenschaften: Typesicherer als Seam, mehr stateful und weniger XML-zentrisch als Spring, mehr Web- und Enterprise-anwendungsfähig als Guice.

Und das Wichtigste - bei Web Beans handelt es sich um einen JCP-Standard, der sich sauber mit Java EE und mit jeder anderen Java SE Umgebung integrieren lässt, bei der einbettbares EJB Lite verfügbar ist.

Erste Schritte mit Web Beans

Können Sie es jetzt kaum erwarten Ihr erstes Web Bean zu schreiben? Oder sind Sie etwas skeptisch und fragen sich, welche Hürden Ihnen bei der Web Beans Spezifikation bevorstehen? Die gute Nachricht ist, dass Sie wahrscheinlich schon hunderte, wenn nicht tausende von Web Beans geschrieben haben. Vielleicht erinnern Sie sich nicht einmal an das erste Web Bean, das Sie je geschrieben haben.

1.1. Ihr erstes Web Bean

Mit bestimmten, ganz besonderen Ausnahmen ist jede Java-Klasse mit einem Konstruktor, die keine Parameter akzeptiert ein Web Bean. Das beinhaltet jedes JavaBean. Desweiteren ist jedes EJB 3-artige Session Bean ein Web Bean. Sicher, die von Ihnen täglich geschriebenen JavaBeans und EJBs konnten die neuen, in der Web Beans Spezifikation definierten Dienste nicht nutzen, aber Sie werden diese allesamt benutzen können können, das Web Beans # diese in andere Web Beans einspeisen, diese via der Web Beans XML-Konfigurationseinrichtung konfigurieren und diesen sogar Interzeptoren und Dekoratoren hinzufügen, ohne den bestehenden Code anzurühren.

Nehmen wir an, Sie besitzen zwei bestehende Java Klassen, die bis dato in verschiedenen Anwendungen verwendet wurden. Die erste Klasse parst einen String in eine Liste von Sätzen:

```
public class SentenceParser {
   public List<String
> parse(String text) { ... }
}
```

Bei der zweiten bestehenden Klasse handelt es sich um das Front-End eines "stateless Session Beans" für ein externes System, das in der Lage ist Sätze von einer Sprache in eine andere zu übersetzen:

```
@Stateless
public class SentenceTranslator implements Translator {
   public String translate(String sentence) { ... }
}
```

Wo `Translator` das lokale Interface ist:

```
@Local
public interface Translator {
   public String translate(String sentence);
```

```
}
```

Leider besitzen wir keine bereits bestehende Klasse die ganze Textdokumente übersetzt. Schreiben wir also ein Web Bean, das diesen Job übernimmt:

```
public class TextTranslator {

   private SentenceParser sentenceParser;
   private Translator sentenceTranslator;

   @Initializer
   TextTranslator(SentenceParser sentenceParser, Translator sentenceTranslator) {
      this.sentenceParser = sentenceParser;
      this.sentenceTranslator = sentenceTranslator;
   }

   public String translate(String text) {
      StringBuilder sb = new StringBuilder();
      for (String sentence: sentenceParser.parse(text)) {
         sb.append(sentenceTranslator.translate(sentence));
      }
      return sb.toString();
   }

}
```

Wir erhalten eine Instanz von `TextTranslator` durch dessen Einspeisung in ein Web Bean, Servlet oder EJB:

```
@Initializer
public setTextTranslator(TextTranslator textTranslator) {
   this.textTranslator = textTranslator;
}
```

Alternativ erhalten wir eine Instanz durch direkten Aufruf einer Methode des Web Bean Managers:

```
TextTranslator tt = manager.getInstanceByType(TextTranslator.class);
```

Aber warten Sie: `TextTranslator` besitzt keinen Konstruktor ohne Parameter! Handelt es sich noch um ein Web Bean? Nun, eine Klasse, die keinen Konstruktor ohne Parameter besitzt, kann nach wie vor Web Bean sein, falls es einen mit `@Initializer` annotierten Konstruktor besitzt.

Wie Sie wahrscheinlich bereits erraten haben, hat die `@Initializer`-Annotation etwas mit Dependency-Einspeisung zu tun! `@Initializer` kann am Konstruktor oder der Methode eines Web Beans angewendet werden und teilt dem Web Bean Manager mit, diesen Konstruktor oder diese Methode bei Instantiierung des Web Beans aufzurufen. Der Web Bean Manager speist andere Web Beans in die Parameter des Konstruktors oder der Methode ein.

Zum Zeitpunkt der Systeminitialisierung muss der Web Bean Manager validieren, dass genau ein Web Bean existiert, das jedem Einspeisungspunkt gerecht wird. Für unser Beispiel bedeutet das, wenn keine Implementierung von `Translator` verfügbar ist # wenn der `SentenceTranslator` EJB nicht deployt wurde # dass der Web Bean Manager eine `UnsatisfiedDependencyException` melden würde. Wäre mehr als eine Implementierung von `Translator` verfügbar, so würde der Web Bean Manager eine `AmbiguousDependencyException` melden.

1.2. Was ist ein Web Bean?

Was also *genau* ist ein Web Bean?

Bei einem Web Bean handelt es sich um eine Anwendungsklasse, die Business Logik enthält. Ein Web Bean kann direkt von Java Code oder via Unified EL aufgerufen werden. Ein Web Bean kann auf transaktionale Ressourcen zugreifen. Abhängigkeiten zwischen Web Beans werden automatisch durch den Web Bean Manager verwaltet. Die meisten Web Beans sind *stateful* und *kontextbezogen*. Der Lebenszyklus eines Web Beans wird immer durch den Web Bean Manager verwaltet.

Erinnern wir uns. Was genau bedeutet es, "kontextuell" zu sein? Da Web Beans "stateful" sein können, ist es relevant *welche* Bean-Instanz ich besitze. Anders als ein Komponentenmodell, das "stateless" ist (etwas "stateless" Session Beans) oder ein Singleton Komponentenmodell (wie Servlets oder Singleton Beans), sehen verschiedene Clients eines Web Beans das Web Bean in unterschiedlichen Stati. Der Client-sichtbare Status ist abhängig davon, auf welche Instanz des Web Beans der Client verweist (eine Referenz besitzt).

Wie beim "stateless" oder "singleton" Modell *anders* jedoch als bei "stateful" Session Beans, steuert der Client den Lebenszyklus der Instanz nicht durch expliziertes Erstellen und Löschen. Stattdessen bestimmt der *Geltungsbereich* des Web Beans:

- der Lebenszyklus jeder Instanz des Web Beans und
- Welche Clients teilen sich eine Referenz zu einer bestimmten Instanz des Web Beans.

Für einen bestimmten Thread in einer Web Beans Anwendung kann ein *aktiver Kontext* mit dem Geltungsbereich des Web Beans assoziiert sein. Dieser Kontext kann eindeutig für den Thread sein (etwa wenn für die Web Bean Anfrage ein Geltungsbereich gilt) oder aber kann mit anderen Threads (etwa wenn für die Web Bean ein Session-Geltungsbereich gilt) oder gar allen Threads (falls ein Anwendungs-Geltungsbereich gilt) geteilt werden.

Clients (etwa andere Web Beans), die in demselben Kontext ausführen sehen dieselbe Instanz des Web Beans. Clients in einem anderen Kontext aber sehen eine andere Instanz.

Ein großer Vorteil des kontextuellen Modells ist es, dass es uns gestattet, stateful Web Beans wie Dienste zu behandeln! Der Client muss sich keine Gedanken um das Management des Lebenszyklus des verwendeten Web Beans machen und *muss nicht einmal wissen was der Lebenszyklus ist.* Web Beans interagieren durch Weitergabe von Nachrichten und die Web Bean Implementierungen definieren den Lebenszyklus ihres eigenen Status. Die Web Beans sind lose gepaart, weil:

- sie interagieren über gut definierte öffentliche APIs
- ihre Lebenszyklen sind vollständig abgekuppelt

Wir können ein Web Bean durch ein anderes Web Bean ersetzen, das dasselbe API und einen anderen Lebenszyklus (einen anderen Geltungsbereich) besitzt, ohne dass die übrige Web Bean Implementierung hiervon betroffen ist. Genau genommen definieren Web Beans eine raffinierte Einrichtung zur Außerkraftsetzung von Web Bean Implementierungen zum Zeitpunkt des Deployment wie wir in *Abschnitt 4.2, „Deployment Typen“* noch sehen werden.

Beachten Sie, dass es sich nicht bei allen Clients eines Web Beans um Web Beans handelt. Andere Objekte wie Servlets oder Message-Driven Beans # die ihrem Wesen nach nicht einspeisbar sind, kontextuelle Objekte # können durch Einspeisung ebenfalls Verweise auf ein Web Beans erhalten.

Formeller gilt, gemäß der Spezifikation:

> Ein Web Bean besteht aus:
>
> - Einem (nicht leeren) Satz von API-Typen
> - Einem (nicht leeren) Satz von bindenden Annotationstypen
> - Einem Geltungsbereich
> - Einem Deployment-Typ
> - Optional einem Web Bean Namen
> - Ein Satz Interzeptor Binding-Typen
> - Einer Web Bean Implementierung

Sehen wir uns jetzt genauer an, was diese Begriffe für einen Entwickler von Web Beans bedeuten.

1.2.1. API-Typen, Binding-Typen und Dependency-Einspeisung

Web Beans erhalten Verweise auf andere Web Beans in der Regel via "Dependency"-Einspeisung. Jedes eingespeiste Attribut legt einen "Vertrag" fest, der vom einzuspeisenden Web Bean erfüllt sein muss. Der Vertrag lautet:

- ein API-Typ, zusammen mit
- einem Satz von Binding-Typen.

Bei einem API handelt es sich um eine benutzerdefinierte Klasse oder Interface. (Falls es sich bei dem Web Bean um ein EJB Session Bean handelt, so ist der API-Typ das `@Local`-Interface oder Bean-Klasse lokale Ansicht). Ein Binding-Typ repräsentiert Client-sichtbare Semantik, die von einigen Implementierungen des API erfüllt wird, von anderen wiederum nicht.

Binding-Typen werden durch benutzerdefinierte Annotationen repräsentiert, die ihrerseits mit `@BindingType` annotiert sind. Zum Beispiel besitzt der folgende Einspeisungspunkt den API-Typ `PaymentProcessor` und Binding-Typ `@CreditCard`:

```
@CreditCard PaymentProcessor paymentProcessor
```

Wird an einem Einspeisungspunkt kein Binding-Typ explizit festgelegt, so wird vom standardmäßigen Binding-Typ `@Current` ausgegangen.

Für jeden Einspeisungspunkt sucht der Web Bean Manager nach einem Web Bean, das den Vertrag erfüllt (das API implementiert und alle Binding-Typen besitzt) und speist dieses Web Bean ein.

Das folgende Web Bean besitzt den Binding-Typ `@CreditCard` und implementiert den API-Typ `PaymentProcessor`. Es könnte daher am Beispiel-Einspeisungspunkt eingespeist werden:

```
@CreditCard
public class CreditCardPaymentProcessor
   implements PaymentProcessor { ... }
```

Falls ein Web Bean nicht explizit einen Satz von Binding-Typen festlegt, so besitzt es genau einen Binding-Typ: den standardmäßigen Binding-Typ `@Current`.

Web Beans definiert einen fortgeschrittenen aber intuitiven *Auflösungsalgorithmus*, der dem Container dabei hilft zu entscheiden was geschehen soll, wenn mehr als ein Web Bean einen bestimmten Vertrag erfüllt. Wir gehen in *Kapitel 4, Dependency-Einspeisung* näher darauf ein.

1.2.2. Deployment-Typen

Deployment-Typen gestatten die Klassifizierung unserer Web Beans mittels Deployment Szenario. Ein Deployment-Typ ist eine Annotation, die ein bestimmtes Deployment-Szenario repräsentiert, etwa `@Mock`, `@Staging` oder `@AustralianTaxLaw`. Wir setzen die Annotation bei Web Beans ein, die in diesem Szenario deployt werden sollten. Ein Deployment-Typ gestattet mit nur einer einzelnen Konfigurationszeile einem ganzen Satz von Web Beans unter Vorbehalt in diesem Szenario deployt zu werden.

Viele Web Beans verwenden nur den standardmäßigen Deployment-Typ `@Production`, in welchem Fall kein Deployment-Typ explizit festgelegt werden muss. Alle drei Web Beans in unserem Beispiel besitzen den Deployment-Typ `@Production`.

In einer Testumgebung können wir das `SentenceTranslator` Web Bean durch ein "mock object" ersetzen:

```
@Mock
public class MockSentenceTranslator implements Translator {
   public String translate(String sentence) {
      return "Lorem ipsum dolor sit amet";
   }
}
```

Wir würden den Deployment-Typ `@Mock` in unserer Testumgebung aktivieren, um anzuzeigen, dass `MockSentenceTranslator` und ein beliebiges anderes mit `@Mock` annotiertes Web Bean verwendet werden sollen.

In *Abschnitt 4.2, „Deployment Typen“* gehen wir näher auf dieses einzigartige und leistungsfähige Feature ein.

1.2.3. Geltungsbereich

Der *Geltungsbereich* definiert den Lebenszyklus und die Sichtbarkeit von Instanzen des Web Beans. Das Web Beans Kontextmodell ist erweiterbar, um arbiträre Geltungsbereiche zu ermöglichen. Jedoch sind bestimmte wichtige Geltungsbereiche in die Spezifikation eingebaut und werden vom Web Bean bereitgestellt. Ein Geltungsbereich wird durch einen Annotationstyp repräsentiert.

Web-Anwendungen können zum Beispiel *Session-begrenzte* Web Beans besitzen:

```
@SessionScoped
public class ShoppingCart { ... }
```

Eine Instanz eines sessionbegrenzten Web Beans wird an eine Benutzer-Session gebunden und wird von allen im Kontext dieser Session ausführenden Anfragen geteilt.

Standardmäßig gehören Web Beans zu einem bestimmten Geltungsbereich namens *abhängiger Pseudo-Geltungsbereich* (sog. "dependent pseudo-scope"). Web Beans mit diesem Geltungsbereich sind reine abhängige Objekte von demjenigen Objekt, in das sie eingespeist werden und ihr Lebenszyklus ist an den Lebenszyklus eben dieses Objekts gebunden.

In *Kapitel 5, Geltungsbereiche und Kontexte* gehen wir näher auf Geltungsbereiche ein.

1.2.4. Web Bean Namen und Unified EL

Ein Web Bean kann einen *Namen* besitzen, wodurch es möglich ist, dieses in Unified EL Ausdrücken einzusetzen. Das Festlegen eines Namens für ein Web Bean ist ganz einfach:

```
@SessionScoped @Named("cart")
public class ShoppingCart { ... }
```

Jetzt können wir das Web Bean einfach in einer beliebigen JSF- oder JSP-Seite verwenden:

```
<h:dataTable value="#{cart.lineItems}" var="item">
    ....
</h:dataTable
>
```

Es ist sogar noch einfacher den Name vom Web Bean Manager standardisieren zu lassen:

```
@SessionScoped @Named
public class ShoppingCart { ... }
```

In diesem Fall wird der Name standardmäßig zu `shoppingCart` # dem nicht vollständigen Klassennamen, wobei der erste Buchstabe klein geschrieben wird.

1.2.5. Interzeptor Binding-Typen

Web Beans unterstützt die von EJB 3 definierte Interzeptor-Funktionalität nicht nur für EJB-Beans, sondern auch für einfache Java-Klassen. Desweiteren bietet Web Beans eine neue Herangehensweise bei der Bindung von Interzeptoren an EJB-Beans und andere Web-Beans.

Es bleibt weiterhin möglich, die Interzeptorklasse mittels Verwendung der `@Interceptors`-Annotation direkt festzulegen:

```
@SessionScoped
@Interceptors(TransactionInterceptor.class)
public class ShoppingCart { ... }
```

Es ist aber eleganter und generell besser das Interzeptor-Binding durch einen *Interzeptor-Binding-Typ* umzuleiten:

```
@SessionScoped @Transactional
public class ShoppingCart { ... }
```

Wir gehen in *Kapitel 7, Interzeptoren* and *Kapitel 8, Dekoratoren* näher auf Web Beans Interzeptoren und Dekoratoren ein.

1.3. Welche Art von Objekten können Web Beans sein?

Wir haben bereits gesehen, dass JavaBeans, EJBs und einige andere Java-Klassen Web Beans sein können. Aber um was für Objekte genau handelt es sich bei Web Beans?

1.3.1. Einfache Web Beans

Die Web Beans Spezifikation besagt, dass eine konkrete Java-Klasse ein *einfaches* Web Bean ist, wenn:

- Es es sich nicht um eine EE Container-gemanagte Komponente wie ein EJB, ein Servlet oder eine JPA-Entity handelt,
- es sich nicht um eine nicht-statische statische innere Klasse handelt,
- es sich nicht um einen parametisierten Typ handelt und
- ein Konstruktor ohne Parameter oder ein mit `@Initializer` annotierter Konstruktor vorhanden ist.

Daher handelt es sich bei fast jedem JavaBean um ein einfaches Web Bean.

Jedes direkt oder indirekt durch ein einfaches Web Bean implementierte Interface ist ein API-Typ des einfachen Web Beans. Die Klasse und deren Superklassen sind ebenfalls API-Typen.

1.3.2. Enterprise Web Beans

Die Spezifikation besagt, dass alle EJB 3-style Session und Singleton Beans *Enterprise* Web Beans sind. Message-driven Beans sind keine Web Beans # da sie nicht zur Einspeisung in andere Objekte vorgesehen sind # aber sie können den größten Teil der Funktionalität von Web Beans nutzen, darunter auch "Dependency"-Einspeisung und Interzeptoren.

Jedes lokale Interface eines Enterprise Web Beans und jedes seiner Super-Interfaces, das keinen Platzhaltertyp-Parameter oder eine Typenvariable besitzt, ist ein API-Typ des Enterprise Web Beans. Falls das EJB-Bean eine lokale Ansicht der Bean-Klasse besitzt, so handelt es sich auch bei der Bean-Klasse und jede von deren Super-Klassen um einen API-Typ.

Stateful Session Beans sollten eine Entfernungsmethode ("remove method") ohne Parameter oder eine Entfernungsmethode mit der Annotation `@Destructor` deklarieren. Der Web Bean Manager ruft diese Methode auf, um die Instanz des stateful Session Beans am Ende von deren Lebenszyklus zu löschen. Diese Methode nennt sich *Destructor*-Methode des Enterprise Web Beans.

```
@Stateful @SessionScoped
public class ShoppingCart {

   ...

   @Remove
   public void destroy() {}

}
```

Sollten wir also ein Enterprise Web Bean statt eines einfachen Web Beans verwenden? Nun, wenn wir ausgefeilte, durch EJB bereitgestellte Enterprise-Dienste benötigen, wie etwa:

- Transaktionsmanagement und Sicherheit auf Methodenebene,
- Nebenläufigkeits-Management,
- Passivation für stateful Session Beans und Instance-Pooling für stateless Session Beans auf Instanzebene
- Remote und Web-Service Aufruf und
- Timer und asynchrone Methoden,

so sollten wir ein Enterprise Web Bean verwenden. Wenn wir nichts von alledem brauchen, so reicht ein einfaches Web Bean vollkommen aus.

Viele Web Beans (einschließlich session- oder anwendungsbegrenzte Web Beans) sind für nebenläufigen Zugriff verfügbar. Daher ist das durch EJB 3.1 bereitgestellte Nebenläufigkeits-Management besonders nützlich. Die meisten session- oder anwendungsbegrenzten Web Beans sollten EJBs sein.

Web Beans, die Verweise auf schwergewichtige Ressourcen oder eine Menge internen Status besitzen, haben Vorteile durch den fortgeschrittenen, Container-gemanagten, durch das EJB `@Stateless`/`@Stateful`/`@Singleton`-Modell definierten Lebenszyklus und dessen Support von Passivation und Instanz-Pooling.

Schließlich ist es offenkundig, wenn Transaktions-Management auf Methodenebene, Sicherheit auf Methodenebene, Timer, Remote-Methoden oder asynchrone Methoden benötigt werden.

Es ist in der Regel leicht, mit einem einfachen Web Bean zu beginnen und es dann zu einem EJB zu machen, indem man eine Annotation: `@Stateless`, `@Stateful` oder `@Singleton` hinzufügt.

1.3.3. Producer-Methoden

Eine *Producer-Methode* ist eine Methode, die vom Web Bean Manager aufgerufen wird, um eine Instanz des Web Beans zu erhalten, wenn im aktuellen Kontext keine existiert. Eine

Producer-Methodübernehmen, statt die Instantiierung dem Web Bean Manager zu überlassen. Zum Beispiel:

```
@ApplicationScoped
public class Generator {

   private Random random = new Random( System.currentTimeMillis() );

   @Produces @Random int next() {
      return random.nextInt(100);
   }

}
```

Einspeisung des Ergebnisses einer Producer-Methode erfolgt wie bei einem regulären Web Bean.

```
@Random int randomNumber
```

Der Methodenwiedergabetyp ("Method Return Type") und alle Interfaces, die er direkt oder indirekt erweitert/implementiert sind API-Typen der Producer-Methode. Handelt es sich beim Wiedergabetyp um eine Klasse, so sind alle Superklassen ebenfalls API-Typen.

Einige Producer-Methoden geben Objekte wieder, die explizite Löschung erfordern:

```
@Produces @RequestScoped Connection connect(User user) {
   return createConnection( user.getId(), user.getPassword() );
}
```

Diese Producer-Methoden können übereinstimmende *Disposal Methods* (Entsorgungsmethoden) definieren:

```
void close(@Disposes Connection connection) {
   connection.close();
}
```

Diese Entsorgungsmethode wird am Ende der Anfrage automatisch vom Web Bean Manager aufgerufen.

In *Kapitel 6, Producer-Methoden* erfahren Sie mehr über Producer-Methoden.

1.3.4. JMS-Endpunkte

Auch eine JMS-Warteschlange oder ein Topic können Web Beans sein. Web Beans nimmt dem Entwickler die Arbeit des Management der Lebenszyklen aller verschiedener JMS-Objekte ab, die zum Senden von Nachrichten an Warteschlangen und Topics erforderlich sind. Wir gehen in *Abschnitt 13.4, „JMS Endpunkte“* auf JMS-Endpunkte ein.

Beispiel einer JSF-Webanwendung

Illustrieren wir diese Ideen an einem vollständigen Beispiel. Wir werden einen Benutzer Login/Logout für eine JSF verwendende Anwendung implementieren. Zunächst definieren wir ein Web Bean das den während des Logins eingegebenen Benutzernamen und das Passwort verwahrt:

```
@Named @RequestScoped
public class Credentials {

   private String username;
   private String password;

   public String getUsername() { return username; }
   public void setUsername(String username) { this.username = username; }

   public String getPassword() { return password; }
   public void setPassword(String password) { this.password = password; }

}
```

Dieses Web Bean ist an den Login-Prompt in folgendem JSF-Formular gebunden:

```
<h:form>
   <h:panelGrid columns="2" rendered="#{!login.loggedIn}">
      <h:outputLabel for="username"
>Username:</h:outputLabel>
      <h:inputText id="username" value="#{credentials.username}"/>
      <h:outputLabel for="password"
>Password:</h:outputLabel>
      <h:inputText id="password" value="#{credentials.password}"/>
   </h:panelGrid>
   <h:commandButton value="Login" action="#{login.login}" rendered="#{!login.loggedIn}"/>
   <h:commandButton value="Logout" acion="#{login.logout}" rendered="#{login.loggedIn}"/>
</h:form
>
```

Die eigentliche Arbeit wird durch ein Session-begrenztes Web Bean übernommen, dass Informationen zum aktuell eingeloggten Benutzer verwahrt und anderen Web Beans die `User`-Entity offenlegt:

```
@SessionScoped @Named
public class Login {

   @Current Credentials credentials;
   @PersistenceContext EntityManager userDatabase;

   private User user;

   public void login() {

      List<User
> results = userDatabase.createQuery(
         "select u from User u where u.username=:username and u.password=:password")
         .setParameter("username", credentials.getUsername())
         .setParameter("password", credentials.getPassword())
         .getResultList();

      if ( !results.isEmpty() ) {
         user = results.get(0);
      }

   }

   public void logout() {
      user = null;
   }

   public boolean isLoggedIn() {
     return user!=null;
   }

   @Produces @LoggedIn User getCurrentUser() {
      return user;
   }

}
```

Natürlich handelt es sich bei `@LoggedIn` um eine "Binding"-Annotation:

```
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD})
@BindingType
```

```
public @interface LoggedIn {}
```

Jetzt kann jedes andere Web Bean den aktuellen Benutzer auf leichte Weise einspeisen:

```
public class DocumentEditor {

    @Current Document document;
    @LoggedIn User currentUser;
    @PersistenceContext EntityManager docDatabase;

    public void save() {
        document.setCreatedBy(currentUser);
        docDatabase.persist(document);
    }

}
```

Wir hoffen, dass Ihnen dieses Beispiel einen Vorgeschmack auf das Web Bean Programmiermodell gegeben hat. Im nächsten Kapitel sehen wir uns die Web Beans Dependency-Einspeisung näher an.

Getting started with Web Beans, the Reference Implementation of JSR-299

The Web Beans is being developed at *the Seam project* [http://seamframework.org/WebBeans]. You can download the latest developer release of Web Beans from the *the downloads page* [http://seamframework.org/Download].

Web Beans comes with a two deployable example applications: `webbeans-numberguess`, a war example, containing only simple beans, and `webbeans-translator` an ear example, containing enterprise beans. There are also two variations on the numberguess example, the tomcat example (suitable for deployment to Tomcat) and the jsf2 example, which you can use if you are running JSF2. To run the examples you'll need the following:

- the latest release of Web Beans,
- JBoss AS 5.0.1.GA, or
- Apache Tomcat 6.0.x, and
- Ant 1.7.0.

3.1. Using JBoss AS 5

You'll need to download JBoss AS 5.0.1.GA from *jboss.org* [http://www.jboss.org/jbossas/downloads/], and unzip it. For example:

```
$ cd /Applications
$ unzip ~/jboss-5.0.1.GA.zip
```

Next, download Web Beans from *seamframework.org* [http://seamframework.org/Download], and unzip it. For example

```
$ cd ~/
$ unzip ~/webbeans-$VERSION.zip
```

Als nächstes müssen wir Web Beans mitteilen, wo JBoss sich befindet. Editieren Sie `jboss-as/build.properties` und setzen Sie die `jboss.home`-Property. Zum Beispiel:

```
jboss.home=/Applications/jboss-5.0.1.GA
```

To install Web Beans, you'll need Ant 1.7.0 installed, and the `ANT_HOME` environment variable set. For example:

Anmerkung

JBoss 5.1.0 comes with Web Beans built in, so there is no need to update the server.

```
$ unzip apache-ant-1.7.0.zip
$ export ANT_HOME=~/apache-ant-1.7.0
```

Then, you can install the update. The update script will use Maven to download Web Beans automatically.

```
$ cd webbeans-$VERSION/jboss-as
$ ant update
```

Jetzt können Sie Ihr erstes Beispiel deployen!

Tipp

The build scripts for the examples offer a number of targets for JBoss AS, these are:

- `ant restart` - Deployment des Beispiels in ausgeklapptem Format
- `ant explode` - Aktualisierung eines ausgeklappten Beispiels ohne Neustart des Deployments
- `ant deploy` - Deployment des Beispiels in komprimiertem jar-Format
- `ant undeploy` - das Beispiel vom Server entfernen
- `ant clean` - Das Beispiel bereinigen

Um das numberguess Beispiel zu deployen:

```
$ cd examples/numberguess
```

```
ant deploy
```

JBoss AS starten:

```
$ /Application/jboss-5.0.0.GA/bin/run.sh
```

Tipp

Falls Sie Windows verwenden, verwenden Sie das `run.bat`-Skript.

Deployen Sie die Anwendung und genießen Sie stundenlangen Spaß unter *http://localhost:8080/webbeans-numberguess*!

Web Beans includes a second simple example that will translate your text into Latin. The numberguess example is a war example, and uses only simple beans; the translator example is an ear example, and includes enterprise beans, packaged in an EJB module. To try it out:

```
$ cd examples/translator
ant deploy
```

Warten Sie, bis die Anwendung deployt ist und besuchen Sie *http://localhost:8080/webbeans-translator*!

3.2. Using Apache Tomcat 6.0

You'll need to download Tomcat 6.0.18 or later from *tomcat.apache.org* [http://tomcat.apache.org/download-60.cgi], and unzip it. For example:

```
$ cd /Applications
$ unzip ~/apache-tomcat-6.0.18.zip
```

Next, download Web Beans from *seamframework.org* [http://seamframework.org/Download], and unzip it. For example

```
$ cd ~/
$ unzip ~/webbeans-$VERSION.zip
```

Next, we need to tell Web Beans where Tomcat is located. Edit `jboss-as/build.properties` and set the `tomcat.home` property. For example:

```
tomcat.home=/Applications/apache-tomcat-6.0.18
```

Tipp

The build scripts for the examples offer a number of targets for Tomcat, these are:

- `ant tomcat.restart` - deploy the example in exploded format
- `ant tomcat.explode` - update an exploded example, without restarting the deployment
- `ant tomcat.deploy` - deploy the example in compressed jar format
- `ant tomcat.undeploy` - remove the example from the server
- `ant tomcat.clean` - clean the example

To deploy the numberguess example for tomcat:

```
$ cd examples/tomcat
ant tomcat.deploy
```

Start Tomcat:

```
$ /Applications/apache-tomcat-6.0.18/bin/startup.sh
```

Tipp

If you use Windows, use the `startup.bat`script.

Deployen Sie die Anwendung und genießen Sie stundenlangen Spaß unter *http://localhost:8080/webbeans-numberguess*!

3.3. Using GlassFish

TODO

3.4. Das numberguess-Beispiel

In der numberguess-Anwendung haben Sie 10 Versuche, eine Zahl zwischen 1 und 100 zu erraten. Nach jedem Versuch wird Ihnen mitgeteilt, ob Sie zu hoch oder zu niedrig liegen.

Das numberguess-Beispiel besteht aus einer Reihe von Web Beans, Konfigurationsdateien und Facelet JSF-Seiten, die als eine war verpackt sind. Fangen wir mit den Konfigurationsdateien an.

Alle Konfigurationsdateien für dieses Beispiel befinden sich in `WEB-INF/`, das in `WebContent` im Quell-Baum gespeichert ist. Zunächst haben wir `faces-config.xml`, in dem wir JSF anweisen, Facelets zu verwenden:

```
<?xml version='1.0' encoding='UTF-8'?>
<faces-config version="1.2"
              xmlns="http://java.sun.com/xml/ns/javaee"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                 xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/
javaee/web-facesconfig_1_2.xsd">

   <application>
      <view-handler
>com.sun.facelets.FaceletViewHandler</view-handler>
   </application>

</faces-config
>
```

Es existiert eine leere `web-beans.xml`-Datei, die diese Anwendung als Web Beans Applikation kennzeichnet.

Und schließlich gibt es noch `web.xml`:

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

<web-app version="2.5"
   xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://java.sun.com/xml/ns/javaee  http://java.sun.com/xml/ns/javaee/
web-app_2_5.xsd">

  <display-name>Web Beans Numbergues example</display-name>

  <!-- JSF -->
```

```
  <servlet>                                                    (1)
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>                                            (2)
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.jsf</url-pattern>
  </servlet-mapping>

  <context-param>                                              (3)
    <param-name>javax.faces.DEFAULT_SUFFIX</param-name>
    <param-value>.xhtml</param-value>
  </context-param>

  <session-config>                                             (4)
    <session-timeout>10</session-timeout>
  </session-config>

</web-app>
```

1. Enable and load the JSF servlet
2. Configure requests to `.jsf` pages to be handled by JSF
3. Tell JSF that we will be giving our source files (facelets) an extension of `.xhtml`
4. Configure a session timeout of 10 minutes

Anmerkung

Whilst this demo is a JSF demo, you can use Web Beans with any Servlet based web framework.

Let's take a look at the Facelet view:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:h="http://java.sun.com/jsf/html"
```

```
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:s="http://jboss.com/products/seam/taglib">


 <ui:composition template="template.xhtml">                    (1)
  <ui:define name="content">
    <h1>Guess a number...</h1>
    <h:form id="NumberGuessMain">

      <div style="color: red">                                  (2)
        <h:messages id="messages" globalOnly="false"/>
          <h:outputText id="Higher" value="Higher!" rendered="#{game.number gt game.guess
and game.guess ne 0}"/>
        <h:outputText id="Lower" value="Lower!" rendered="#{game.number lt game.guess and
game.guess ne 0}"/>
      </div>

      <div>

        I'm thinking of a number between #{game.smallest} and #{game.bigge(3)st}.
        You have #{game.remainingGuesses} guesses.
      </div>

      <div>
        Your guess:

        <h:inputText id="inputGuess"                            (4)
                     value="#{game.guess}"
                     required="true"
                     size="3"
                     disabled="#{game.number eq game.guess}">

          <f:validateLongRange maximum="#{game.biggest}"        (5)
                               minimum="#{game.smallest}"/>
        </h:inputText>

       <h:commandButton id="GuessButton"                        (6)
                        value="Guess"
                        action="#{game.check}"
                        disabled="#{game.number eq game.guess}"/>
      </div>
      <div>
                    <h:commandButton  id="RestartButton"  value="Reset"  action="#{game.reset}"
immediate="true" />
      </div>
    </h:form>
```

```
   </ui:define>
 </ui:composition>
</html>
```

1. Facelets is a templating language for JSF, here we are wrapping our page in a template which defines the header.
2. There are a number of messages which can be sent to the user, "Higher!", "Lower!" and "Correct!"
3. As the user guesses, the range of numbers they can guess gets smaller - this sentance changes to make sure they know what range to guess in.
4. This input field is bound to a Web Bean, using the value expression.
5. A range validator is used to make sure the user doesn't accidentally input a number outside of the range in which they can guess - if the validator wasn't here, the user might use up a guess on an out of range number.
6. And, of course, there must be a way for the user to send their guess to the server. Here we bind to an action method on the Web Bean.

Das Beispiel besteht aus 4 Klassen, wobei die ersten beiden Binding-Typen sind. Zunächst gibt es den `@Random` Binding-Typ, der zur Einspeisung einer zufälligen Zahl dient:

```
@Target( { TYPE, METHOD, PARAMETER, FIELD })
@Retention(RUNTIME)
@Documented
@BindingType
public @interface Random {}
```

Es gibt außerdem den `@MaxNumber` Binding-Typ, der zur Einspeisung der maximalen Zahl, die eingespeist werden kann, verwendet wird:

```
@Target( { TYPE, METHOD, PARAMETER, FIELD })
@Retention(RUNTIME)
@Documented
@BindingType
public @interface MaxNumber {}
```

Die `Generator`-Klasse ist verantwortlich für die Erstellung der zufälligen Zahl via einer Producer-Methode. Sie legt auch die mögliche Maximalzahl via einer maximum Producer-Methode offen:

```
@ApplicationScoped
public class Generator {
```

```
  private java.util.Random random = new java.util.Random( System.currentTimeMillis() );

  private int maxNumber = 100;

  java.util.Random getRandom()
  {
    return random;
  }

  @Produces @Random int next() {
    return getRandom().nextInt(maxNumber);
  }

  @Produces @MaxNumber int getMaxNumber()
  {
    return maxNumber;
  }

}
```

Sie werden feststellen, dass der `Generator` anwendungsbegrenzt ist; daher erhalten wir nicht jedes Mal ein anderes Zufallsergebnis.

Das letzte Web Bean in der Anwendung ist das sessionbegrenzte `Game`.

Sie werden bemerken, dass wir die `@Named`-Annotation verwendet haben, damit wir das Bean durch EL in der JSF-Seite verwenden können. Zu guter Letzt haben wir Konstruktor-Einspeisung zur Initialisierung des Spiels mit Zufallszahl verwendet. Und natürlich müssen wir dem Spieler mitteilen, wenn er gewonnen hat, daher bieten wir Feedback mittels einer `FacesMessage`.

```
package org.jboss.webbeans.examples.numberguess;


import javax.annotation.PostConstruct;
import javax.faces.application.FacesMessage;
import javax.faces.context.FacesContext;
import javax.webbeans.AnnotationLiteral;
import javax.webbeans.Current;
import javax.webbeans.Initializer;
import javax.webbeans.Named;
import javax.webbeans.SessionScoped;
import javax.webbeans.manager.Manager;

@Named
```

```
@SessionScoped
public class Game
{
   private int number;

   private int guess;
   private int smallest;
   private int biggest;
   private int remainingGuesses;

   @Current Manager manager;

   public Game()
   {
   }

   @Initializer
   Game(@MaxNumber int maxNumber)
   {
      this.biggest = maxNumber;
   }

   public int getNumber()
   {
      return number;
   }

   public int getGuess()
   {
      return guess;
   }

   public void setGuess(int guess)
   {
      this.guess = guess;
   }

   public int getSmallest()
   {
      return smallest;
   }

   public int getBiggest()
   {
```

```
    return biggest;
  }

  public int getRemainingGuesses()
  {
    return remainingGuesses;
  }

  public String check()
  {
    if (guess
>number)
    {
      biggest = guess - 1;
    }
    if (guess<number)
    {
      smallest = guess + 1;
    }
    if (guess == number)
    {
      FacesContext.getCurrentInstance().addMessage(null, new FacesMessage("Correct!"));
    }
    remainingGuesses--;
    return null;
  }

  @PostConstruct
  public void reset()
  {
    this.smallest = 0;
    this.guess = 0;
    this.remainingGuesses = 10;
    this.number = manager.getInstanceByType(Integer.class, new AnnotationLiteral<Random
>(){});
  }

}
```

3.4.1. The numberguess example in Tomcat

The numberguess for Tomcat differs in a couple of ways. Firstly, Web Beans should be deployed as a Web Application library in `WEB-INF/lib`. For your convenience we provide a single jar suitable for running Web Beans in any servlet container `webbeans-servlet.jar`.

Tipp

Of course, you must also include JSF and EL, as well common annotations (`jsr250-api.jar`) which a JEE server includes by default.

Secondly, we need to explicitly specify the servlet listener (used to boot Web Beans, and control it's interaction with requests) in `web.xml`:

```
<listener>
  <listener-class>org.jboss.webbeans.environment.servlet.Listener</listener-class>
</listener>
```

3.4.2. The numberguess example for Apache Wicket

Whilst JSR-299 specifies integration with Java ServerFaces, Web Beans allows you to inject into Wicket components, and also allows you to use a conversation context with Wicket. In this section, we'll walk you through the Wicket version of the numberguess example.

Anmerkung

You may want to review the Wicket documentation at *http://wicket.apache.org/*.

Like the previous example, the Wicket WebBeans examples make use of the `webbeans-servlet` module. The use of the *Jetty servlet container* [http://jetty.mortbay.org/] is common in the Wicket community, and is chosen here as the runtime container in order to facilitate comparison between the standard Wicket examples and these examples, and also to show how the webbeans-servlet integration is not dependent upon Tomcat as the servlet container.

These examples make use of the Eclipse IDE; instructions are also given to deploy the application from the command line.

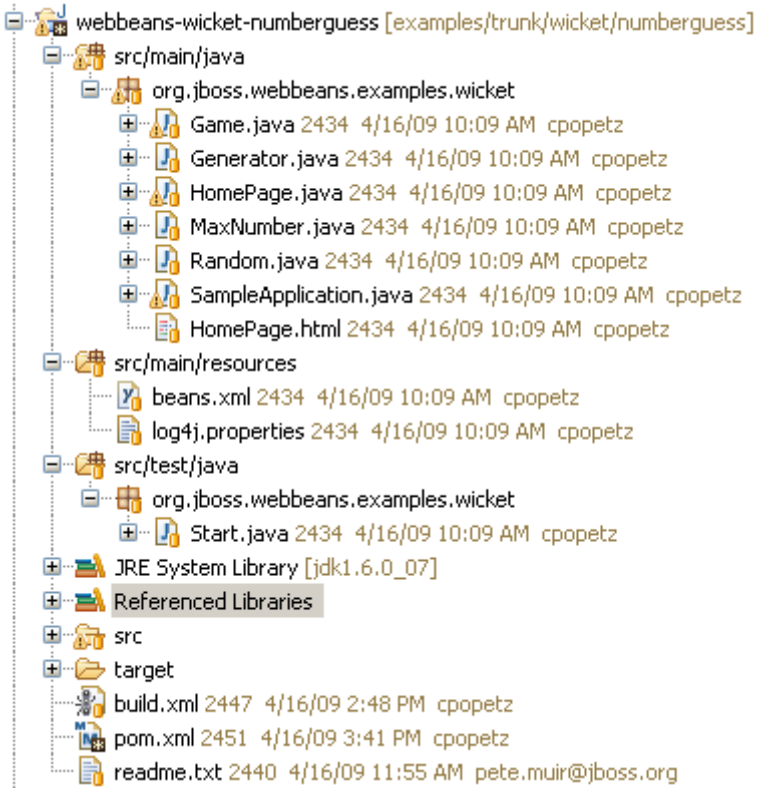
3.4.2.1. Creating the Eclipse project

To generate an Eclipse project from the example:

```
cd examples/wicket/numberguess
mvn -Pjetty eclipse:eclipse
```

Then, from eclipse, choose *File -> Import -> General -> Existing Projects into Workspace*, select the root directory of the numberguess example, and click finish. Note that if you do not intend to

run the example with jetty from within eclipse, omit the "-Pjetty." This will create a project in your workspace called `webbeans-wicket-numberguess`



3.4.2.2. Running the example from Eclipse

This project follows the `wicket-quickstart` approach of creating an instance of Jetty in the `Start` class. So running the example is as simple as right-clicking on that Start class in `src/test/java` in the *Package Explorer* and choosing *Run as Java Application*. You should see console output related to Jetty starting up; then visit able `http://localhost:8080` to view the app. To debug choose *Debug as Java Application*.

3.4.2.3. Running the example from the command line in JBoss AS or Tomcat

This example can also be deployed from the command line in a (similar to the other examples). Assuming you have set up the `build.properties` file in the `examples` directory to specify the location of JBoss AS or Tomcat, as previously described, you can run `ant deploy` from the `examples/wicket/numberguess` directory, and access the application at `http://localhost:8080/webbeans-numberguess-wicket`.

3.4.2.4. Understanding the code

JSF uses Unified EL expressions to bind view layer components in JSP or Facelet views to beans, Wicket defines it's components in Java. The markup is plain html with a one-to-one mapping between html elements and the view components. All view logic, including binding of components to models and controlling the response of view actions, is handled in Java. The integration of

Web Beans with Wicket takes advantage of the same binding annotations used in your business layer to provide injection into your WebPage subclass (or into other custom wicket component subclasses).

The code in the wicket numberguess example is very similar to the JSF-based numberguess example. The business layer is identical!

Differences are:

- Each wicket application must have a `WebApplication` subclass, In our case, our application class is `SampleApplication`:

```
public class SampleApplication extends WebBeansApplication {
  @Override
  public Class getHomePage() {
    return HomePage.class;
  }
}
```

  This class specifies which page wicket should treat as our home page, in our case, `HomePage.class`

- In `HomePage` we see typical wicket code to set up page elements. The bit that is interesting is the injection of the `Game` bean:

```
@Current Game game;
```

  The `Game` bean is can then be used, for example, by the code for submitting a guess:

```
final Component guessButton = new AjaxButton("GuessButton") {
  protected void onSubmit(AjaxRequestTarget target, Form form) {
    if (game.check()) {
```

Anmerkung

All injections may be serialized; actual storage of the bean is managed by JSR-299. Note that Wicket components, like the HomePage and it subcomponents, are *not* JSR-299 beans.

Wicket components allow injection, but they *cannot* use interceptors, decorators and lifecycle callbacks such as `@PostConstruct` or `@Initializer` methods.

- The example uses AJAX for processing of button events, and dynamically hides buttons that are no longer relevant, for example when the user has won the game.
- In order to activate wicket for this webapp, the Wicket filter is added to web.xml, and our application class is specified:

```
<filter>
  <filter-name>wicket.numberguess-example</filter-name>
  <filter-class>org.apache.wicket.protocol.http.WicketFilter</filter-class>
  <init-param>
    <param-name>applicationClassName</param-name>
    <param-value>org.jboss.webbeans.examples.wicket.SampleApplication</param-value>
  </init-param>
</filter>

<filter-mapping>
  <filter-name>wicket.numberguess-example</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>

<listener>
  <listener-class>org.jboss.webbeans.environment.servlet.Listener</listener-class>
</listener>
```

Note that the servlet listener is also added, as in the Tomcat example, in order to boostrap Web Beans when Jetty starts, and to hook Web Beans into the Jetty servlet request and session lifecycles.

3.4.3. The numberguess example for Java SE with Swing

This example can be found in the `examples/se/numberguess` folder of the Web Beans distribution.

To run this example:

- Open a command line/terminal window in the `examples/se/numberguess` directory
- Ensure that Maven 2 is installed and in your PATH
- Ensure that the `JAVA_HOME` environment variable is pointing to your JDK installation
- execute the following command

```
mvn -Drun
```

There is an empty `beans.xml` file in the root package (`src/main/resources/beans.xml`), which marks this application as a Web Beans application.

The game's main logic is located in `Game.java`. Here is the code for that class, highlighting the changes made from the web application version:

```
@ApplicationScoped                                                    (1) (2)
public class Game implements Serializable
{

   private int number;
   private int guess;
   private int smallest;

   @MaxNumber
   private int maxNumber;

   private int biggest;
   private int remainingGuesses;
   private boolean validNumberRange = true;

   @Current Generator rndGenerator;

   ...

   public boolean isValidNumberRange()
   {
      return validNumberRange;
   }

   public boolean isGameWon()
                                                                      (3)
   {
      return guess == number;
   }

   public boolean isGameLost()
   {
      return guess != number && remainingGuesses <= 0;
   }

   public boolean check()
   {
      boolean result = false;
```

```
        if ( checkNewNumberRangeIsValid() )                    4
        {
            if ( guess > number )
            {
                biggest = guess - 1;
            }

            if ( guess < number )
            {
                smallest = guess + 1;
            }

            if ( guess == number )
            {
                result = true;
            }

            remainingGuesses--;
        }

        return result;
    }

    private boolean checkNewNumberRangeIsValid()
    {
        return validNumberRange = ( ( guess >= smallest ) && ( guess <= biggest ) );
    }

    @PostConstruct
    public void reset()                                        5
    {
        this.smallest = 0;
        ...
        this.number = rndGenerator.next();
    }
}
```

1. The bean is application scoped instead of session scoped, since an instance of the application represents a single 'session'.
2. The bean is not named, since it doesn't need to be accessed via EL

3. There is no JSF `FacesContext` to add messages to. Instead the `Game` class provides additional information about the state of the current game including:

   - If the game has been won or lost
   - If the most recent guess was invalid

   This allows the Swing UI to query the state of the game, which it does indirectly via a class called `MessageGenerator`, in order to determine the appropriate messages to display to the user during the game.
4. Validation of user input is performed during the `check()` method, since there is no dedicated validation phase
5. The `reset()` method makes a call to the injected `rndGenerator` in order to get the random number at the start of each game. It cannot use `manager.getInstanceByType(Integer.class, new AnnotationLiteral<Random>(){})` as the JSF example does because there will not be any active contexts like there is during a JSF request.

The `MessageGenerator` class depends on the current instance of `Game`, and queries its state in order to determine the appropriate messages to provide as the prompt for the user's next guess and the response to the previous guess. The code for `MessageGenerator` is as follows:

```
public class MessageGenerator
{
   @Current Game game;                                                    1


   public String getChallengeMessage()                                    2
   {
      StringBuilder challengeMsg = new StringBuilder( "I'm thinking of a number between " );
      challengeMsg.append( game.getSmallest() );
      challengeMsg.append( " and " );
      challengeMsg.append( game.getBiggest() );
      challengeMsg.append( ". Can you guess what it is?" );

      return challengeMsg.toString();
   }


   public String getResultMessage()                                       3
   {
      if ( game.isGameWon() )
      {
         return "You guess it! The number was " + game.getNumber();
```

```
        } else if ( game.isGameLost() )
        {
            return "You are fail! The number was " + game.getNumber();
        } else if ( ! game.isValidNumberRange() )
        {
            return "Invalid number range!";
        } else if ( game.getRemainingGuesses() == Game.MAX_NUM_GUESSES )
        {
            return "What is your first guess?";
        } else
        {
            String direction = null;

            if ( game.getGuess() < game.getNumber() )
            {
                direction = "Higher";
            } else
            {
                direction = "Lower";
            }

            return direction + "! You have " + game.getRemainingGuesses() + " guesses left.";
        }
    }
}
```

1. The instance of `Game` for the application is injected here.
2. The `Game`'s state is interrogated to determine the appropriate challenge message.
3. And again to determine whether to congratulate, console or encourage the user to continue.

Finally we come to the `NumberGuessFrame` class which provides the Swing front end to our guessing game.

```
public class NumberGuessFrame  extends javax.swing.JFrame
{
    private @Current Game game;                                    (1)

    private @Current MessageGenerator msgGenerator;                (2)


    public void start( @Observes @Deployed Manager manager )       (3)
    {
        java.awt.EventQueue.invokeLater( new Runnable()
```

```
        {
            public void run()
            {
                initComponents();
                setVisible( true );
            }
        } );
    }

    private void initComponents() {                                    4

        buttonPanel = new javax.swing.JPanel();
        mainMsgPanel = new javax.swing.JPanel();
        mainLabel = new javax.swing.JLabel();
        messageLabel = new javax.swing.JLabel();
        guessText = new javax.swing.JTextField();
        ...
        mainLabel.setText(msgGenerator.getChallengeMessage());
        mainMsgPanel.add(mainLabel);

        messageLabel.setText(msgGenerator.getResultMessage());
        mainMsgPanel.add(messageLabel);
        ...
    }

    private void guessButtonActionPerformed( java.awt.event.ActionEvent evt )   5
    {
        int guess =  Integer.parseInt(guessText.getText());

        game.setGuess( guess );
        game.check();
        refreshUI();

    }

    private void replayBtnActionPerformed( java.awt.event.ActionEvent evt )   6
    {
      game.reset();
      refreshUI();
    }

    private void refreshUI()                                           7
```

```
    {
        mainLabel.setText( msgGenerator.getChallengeMessage() );
        messageLabel.setText( msgGenerator.getResultMessage() );
        guessText.setText( "" );
        guessesLeftBar.setValue( game.getRemainingGuesses() );
        guessText.requestFocus();
    }

    // swing components
    private javax.swing.JPanel borderPanel;
    ...
    private javax.swing.JButton replayBtn;

}
```

1. The injected instance of the game (logic and state).
2. The injected message generator for UI messages.
3. This application is started in the usual Web Beans SE way, by observing the `@Deployed Manager` event.
4. This method initialises all of the Swing components. Note the use of the `msgGenerator`.
5. `guessButtonActionPerformed` is called when the 'Guess' button is clicked, and it does the following:

    - Gets the guess entered by the user and sets it as the current guess in the `Game`
    - Calls `game.check()` to validate and perform one 'turn' of the game
    - Calls `refreshUI`. If there were validation errors with the input, this will have been captured during `game.check()` and as such will be reflected in the messeges returned by `MessageGenerator` and subsequently presented to the user. If there are no validation errors then the user will be told to guess again (higher or lower) or that the game has ended either in a win (correct guess) or a loss (ran out of guesses).
6. `replayBtnActionPerformed` simply calls `game.reset()` to start a new game and refreshes the messages in the UI.
7. `refreshUI` uses the `MessageGenerator` to update the messages to the user based on the current state of the Game.

3.5. Das translator-Beispiel

Beim translator-Beispiel werden die von Ihnen eingegebenen Sätze ins Lateinische übersetzt.

Das translator-Beispiel ist eine ear und enthält EJBs. Als Folge ist seine Struktur komplexer als die desnumberguess-Beispiels.

Anmerkung

EJB 3.1 und Jave EE 6 gestatten es Ihnen EJBs in eine war zu verpacken, wodurch diese Struktur wesentlich einfacher wird!

Werfen wir zunächst einen Blick auf den ear-Aggregator, das sich im `webbeans-translator-ear`-Modul befindet. Maven generiert automatisch die `application.xml` für uns:

```
<plugin>
  <groupId
>org.apache.maven.plugins</groupId>
  <artifactId
>maven-ear-plugin</artifactId>
  <configuration>
    <modules>
      <webModule>
        <groupId
>org.jboss.webbeans.examples.translator</groupId>
        <artifactId
>webbeans-translator-war</artifactId>
        <contextRoot
>/webbeans-translator</contextRoot>
      </webModule>
    </modules>
  </configuration>
</plugin
>
```

Hier setzen wir den Kontextpfad, der uns eine schöne url liefert (*http://localhost:8080/webbeans-translator*).

Tipp

Falls Sie zur Generierung dieser Dateien nicht Maven verwendet haben, benötigen Sie `META-INF/application.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<application xmlns="http://java.sun.com/xml/ns/javaee"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                 xsi:schemaLocation="http://java.sun.com/xml/ns/javaee  http://
java.sun.com/xml/ns/javaee/application_5.xsd"
```

```
            version="5">
  <display-name
>webbeans-translator-ear</display-name>
  <description
>Ear Example for the reference implementation of JSR 299: Web Beans</
description>

  <module>
    <web>
      <web-uri
>webbeans-translator.war</web-uri>
      <context-root
>/webbeans-translator</context-root>
    </web>
  </module>
  <module>
    <ejb
>webbeans-translator.jar</ejb>
  </module>
</application
>
```

Next, lets look at the war. Just as in the numberguess example, we have a `faces-config.xml` (to enable Facelets) and a `web.xml` (to enable JSF) in `WebContent/WEB-INF`.

Interessanter ist das zur Übersetzung des Texts verwendete Facelet. Ganz wie im numberguess-Beispiel besitzen wir eine Vorlage, die das Formular umgibt (hier der Kürze wegen weggelassen):

```
<h:form id="NumberGuessMain">

  <table>
    <tr align="center" style="font-weight: bold" >
      <td>
        Your text
      </td>
      <td>
        Translation
      </td>
    </tr>
    <tr>
      <td>
      <h:inputTextarea id="text" value="#{translator.text}" required="true" rows="5" cols="80" />
      </td>
```

```
        <td>
          <h:outputText value="#{translator.translatedText}" />
        </td>
      </tr>
    </table>
    <div>
      <h:commandButton id="button" value="Translate" action="#{translator.translate}"/>
    </div>

</h:form
>
```

Der Benutzer kann Text im Textbereich links eingeben und dann die translate-Schaltfläche drücken (zur Übersetzung), um auf der rechten Seite das Ergebnis zu sehen.

Sehen wir uns schließlich noch das ejb-Modul `webbeans-translator-ejb` an. In `src/main/resources/META-INF` existiert nur eine leere `web-beans.xml`, die dazu dient das Archiv als Web Beans enthaltend zu markieren.

Wir haben uns das Interessanteste bis zuletzt aufgehoben, nämlich den Code! Das Projekt besitzt zwei einfache Beans, `SentenceParser` und `TextTranslator` und zwei Enterprise Beans, `TranslatorControllerBean` und `SentenceTranslator`. Sie sind wahrscheinlich schon weitehend vertraut damit, wie Web Bean aussehen, daher gehen wir hier nur auf die interessantesten Aspekte ein.

Sowohl bei `SentenceParser` als auch bei `TextTranslator` handelt es sich um abhängige Beans und `TextTranslator` verwendet Konstruktor-Initialisierung :

```
public class TextTranslator {
  private SentenceParser sentenceParser;
  private Translator sentenceTranslator;

  @Initializer
  TextTranslator(SentenceParser sentenceParser, Translator sentenceTranslator)
  {
    this.sentenceParser = sentenceParser;
    this.sentenceTranslator = sentenceTranslator;
```

`TextTranslator` ist ein stateless Bean (mit einem lokalen Business-Interface), wo alles passiert - natürlich konnten wir keinen kompletten Übersetzer entwickeln,, aber wir haben uns Mühe gegeben!

Schließlich gibt es noch den UI-orientierten Kontroller, der den Text vom Benutzer nimmt und ihn an den translator (Übersetzer) weitergibt. Hierbei handelt es sich um ein anfragenbegrenztes, benanntes, stateful Session Bean, das den translator einspeist.

```
@Stateful
@RequestScoped
@Named("translator")
public class TranslatorControllerBean implements TranslatorController
{

   @Current TextTranslator translator;
```

Das Bean besitzt auch Getter und Setter für alle Felder auf der Seite.

Da es sich um ein stateful Session Bean handelt, müssen wir eine remove-Methode besitzen:

```
   @Remove
   public void remove()
   {

   }
```

Der Web Beans Manager ruft die remove-Methode für Sie auf, wenn das Bean gelöscht wird, in diesem Fall am Ende der Anfrage.

That concludes our short tour of the Web Beans examples. For more on Web Beans , or to help out, please visit *http://www.seamframework.org/WebBeans/Development*.

Wir brauche Unterstützung auf allen Gebieten - Fehlerbehebung, Schreiben neuer Features, Schreiben neuer Beispiele und bei der Übersetzung dieses Referenzhandbuchs.

Dependency-Einspeisung

Web Beans unterstützt drei primäre Mechanismen für "Dependency"-Einspeisung:

Konstruktorparameter-Einspeisung:

```
public class Checkout {

   private final ShoppingCart cart;

   @Initializer
   public Checkout(ShoppingCart cart) {
      this.cart = cart;
   }

}
```

Initializer-Methode Parameter-Einspeisung:

```
public class Checkout {

   private ShoppingCart cart;

   @Initializer
   void setShoppingCart(ShoppingCart cart) {
      this.cart = cart;
   }

}
```

Und direkte Feldeinspeisung:

```
public class Checkout {

   private @Current ShoppingCart cart;

}
```

Dependency-Einspeisung findet stets bei der ersten Instantiierung der Web Bean Instanz statt.

- Zuerst ruft der Web Bean Manager den Web Bean Konstruktor auf, um eine Instanz des Web Beans zu erhalten.
- Als nächstes initialisiert der Web Bean Manager die Werte aller eingespeisten Felder des Web Beans.
- Anschließend ruft der Web Bean Manager alle Initialisierermethoden des Web Beans auf.
- Zuletzt wird die `@PostConstruct`-Methode des Web Bean, falls vorhanden, aufgerufen.

Die Einspeisung von Konstruktorparametern wird für EJB-Beans nicht unterstützt, da das EJB durch den EJB-Container und nicht den Web Bean Manager instantiiert wird.

Parameter von Konstruktoren und Initialisierermethoden müssen nicht explizit annotiert werden, wenn der standardmäßige Binding-Typ `@Current` gilt.Eingespeiste Felder jedoch *müssen* einen Binding-Typ festlegen, selbst wenn der standardmäßige Binding-Typ gilt. Legt das Feld keinen standardmäßige Binding-Typ fest, so wird es nicht eingespeist.

Producer-Methoden unterstützen Parametereinspeisung ebenfalls:

```
@Produces Checkout createCheckout(ShoppingCart cart) {
   return new Checkout(cart);
}
```

Observer-Methoden (auf die wir in *Kapitel 9, Ereignisse* näher eingehen), "Disposal"-Methoden und "Destructor"-Methoden unterstützen allesamt die Parametereinspeisung.

Die Web Beans Spezifikation definiert eine Prozedur namens *typesicherer Auflösungsalgorithmus* (sog. typesafe resolution algorithm), den der Web Bean Manager bei der Identifizierung des an einem Einspeisungspunkt einzuspeisenden Web Beans folgt. Dieser Algorithmus sieht auf den ersten Blick recht komplex aus, ist es aber nach kurzer Eingewöhnung nicht. Typensichere Auflösung wird zum Initialisierungszeitpunkt des Systems durchgeführt, was bedeutet, dass der Manager den Benutzer sofort darüber informiert, falls die Abhängigkeiten eines Web Beans nicht erfüllt werden können - dies erfolgt durch Meldung von `UnsatisfiedDependencyException` oder `AmbiguousDependencyException`.

Der Zweck dieses Algorithmus ist es, mehreren Web Beans die Einspeisung desselben API-Typs zu gestatten und entweder:

- Dem Client mittels *Binding-Annotationen* zu gestatten auszuwählen, welche Implemetierung er benötigt,
- Dem Anwendungs-Deployer durch Aktivierung oder Deaktivierung von *Deployment-Typen* gestatten auszuwählen, welche Implementierung die passende für eine bestimmte Umgebung ist, ohne dass es zu Änderungen am Client kommt oder

- Einer Implementierung eines API mittels *Deployment-Typ Präzedenz* ("Deployment Type Precedence") gestatten, zum Deployment-Zeitpunkt eine andere Implementerung desselben API außer Kraft zu setzen, ohne dass dies zu Änderungen am Client führt..

Schauen wir uns jetzt näher an, wie der Web Beans Manager ein einzuspeisendes Web Bean bestimmt.

4.1. Binding-Annotationen

Falls mehr als ein Web Bean existiert, das einen bestimmten API-Typ implementiert, so kann der Einspeisungspunkt genau festlegen welches Web Bean eingespeist wird mittels Binding-Annotation. Zum Beispiel können zwei Implementierungen von `PaymentProcessor` vorhanden sein:

```
@PayByCheque
public class ChequePaymentProcessor implements PaymentProcessor {
   public void process(Payment payment) { ... }
}
```

```
@PayByCreditCard
public class CreditCardPaymentProcessor implements PaymentProcessor {
   public void process(Payment payment) { ... }
}
```

Wo `@PayByCheque` und `@PayByCreditCard` Binding-Annotationen sind:

```
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER})
@BindingType
public @interface PayByCheque {}
```

```
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER})
@BindingType
public @interface PayByCreditCard {}
```

Ein Client Web Bean Entwickler verwendet die Binding-Annotation um genau festzulegen, welches Web Bean eingespeist werden sollte.

Verwendung der Feldeinspeisung:

```
@PayByCheque PaymentProcessor chequePaymentProcessor;
@PayByCreditCard PaymentProcessor creditCardPaymentProcessor;
```

Verwendung der Initialisierermethoden-Einspeisung:

```
@Initializer
public void setPaymentProcessors(@PayByCheque PaymentProcessor
 chequePaymentProcessor,
                    @PayByCreditCard PaymentProcessor creditCardPaymentProcessor) {
  this.chequePaymentProcessor = chequePaymentProcessor;
  this.creditCardPaymentProcessor = creditCardPaymentProcessor;
}
```

Oder Verwendung der Konstruktoreinspeisung:

```
@Initializer
public Checkout(@PayByCheque PaymentProcessor chequePaymentProcessor,
          @PayByCreditCard PaymentProcessor creditCardPaymentProcessor) {
  this.chequePaymentProcessor = chequePaymentProcessor;
  this.creditCardPaymentProcessor = creditCardPaymentProcessor;
}
```

4.1.1. Binding-Annotationen mit Mitgliedern

Binding-Annotationen können Mitglieder besitzen:

```
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER})
@BindingType
public @interface PayBy {
   PaymentType value();
}
```

Wobei in diesem Fall der Mitgliederwert von Bedeutung ist:

```
@PayBy(CHEQUE) PaymentProcessor chequePaymentProcessor;
@PayBy(CREDIT_CARD) PaymentProcessor creditCardPaymentProcessor;
```

Sie können den Web Bean Manager anweisen, ein Mitglied eines Binding-Annotationstyps zu ignorieren, indem Sie das Mitglied mit `@NonBinding` annotieren.

4.1.2. Kombinationen von Binding-Annnotationen

Ein Einspeisungspunkt kann sogar mehrere Binding-Annotationen festlegen:

```
@Asynchronous @PayByCheque PaymentProcessor paymentProcessor
```

In diesem Fall würde nur ein Web Bean mit *beiden* Binding-Annotationen eingespeist.

4.1.3. Binding-Annotationen und Producer-Methoden

Sogar Producer-Methoden können Binding-Annotationen festlegen:

```
@Produces
@Asynchronous @PayByCheque
PaymentProcessor createAsyncPaymentProcessor(@PayByCheque PaymentProcessor
 processor) {
   return new AsynchronousPaymentProcessor(processor);
}
```

4.1.4. Der standardmäßige Binding-Typ

Web Beans definiert einen Binding-Typ `@Current`, der der standardmäßige Binding-Typ für jeden Einspeisungspunkt oder Web Bean ist, der nicht explizit einen Binding-Typ festlegt.

Es existieren zwei gängige Umstände, bei denen es notwendig ist, `@Current` festzulegen:

- An einem Feld, um es als eingespeistes Feld zu deklarieren mit dem standardmäßigen Binding-Typ und
- an einem Web Bean, das neben dem standardmäßigen Binding-Type einen weiteren Binding-Typ besitzt.

4.2. Deployment Typen

Alle Web Beans besitzen einen *Deployment-Typ*. Jeder Deployment-Typ identifiziert einen Satz von Web Beans mit Vorbehalt in einigen Deployments des Systems installiert werden sollten.

Zum Beispiel könnten wir einen Deployment-Typ namens `@Mock` definieren, der Web Beans identifiziert, die nur installiert werden sollen, wenn das System innerhalb einer Integrationstestumgebung ausgeführt wird:

```
@Retention(RUNTIME)
 @Target({TYPE, METHOD})
 @DeploymentType
 public @interface Mock {}
```

Nehmen wir an, wir hätten ein Web Bean, das mit einem externen System interagiert, um Zahlungen zu bearbeiten:

```
public class ExternalPaymentProcessor {

   public void process(Payment p) {
      ...
   }

}
```

Da dieses Web Bean nicht explizit einen Deployment-Typ festlegt gilt der standardmäßige Deployment-Typ `@Production`.

Für Integration oder das Testen von Einheiten ist das externe System langsam oder nicht verfügbar. Daher würden wir ein "Mock"-Objekt erstellen:

```
@Mock
public class MockPaymentProcessor implements PaymentProcessor {

   @Override
   public void process(Payment p) {
      p.setSuccessful(true);
   }

}
```

Wie aber bestimmt der Web Bean Manager, welche Implementierung in einem bestimmten Deployment verwendet werden soll?

4.2.1. Aktivierung von Deployment-Typen

Web Beans definieren zwei eingebaute Deployment-Typen: `@Production` und `@Standard`. Standardmäßig sind nur Web Beans mit den eingebauten Deployment-Typen bei Deployment des Systems aktiviert. Wir können weitere Deployment-Typen identifizieren, die bei einem bestimmten Deployment aktiviert werden sollen, indem wir diese in `web-beans.xml` aufführen.

Kehren wir zu unserem Beispiel zurück, wenn wir Integrationsstests deployen und wir möchten, dass alle unsere `@Mock`-Objekte installiert werden:

```
<WebBeans>
   <Deploy>
      <Standard/>
      <Production/>
      <test:Mock/>
   </Deploy>
</WebBeans
>
```

Jetzt identifiziert und installiert der Web Bean Manager alle mit `@Production`, `@Standard` oder `@Mock` annotierten Web Beans zum Zeitpunkt des Deployments.

Der Deployment-Typ `@Standard` wird nur für bestimmte, spezielle durch die Web Beans Spezifikation definierte Web Beans verwendet. Wir können ihn nicht für unsere eigenen Web Beans benutzen und wir können ihn nicht deaktivieren.

Der Deployment-Typ `@Production` ist der standardmäßige Deployment-Typ für Web Beans, die keinen expliziten Deployment-Typ festlegen oder deaktiviert sind.

4.2.2. Deployment-Typ Präzedenz

Wenn Sie aufgepasst haben, fragen Sie sich jetzt wahrscheinlich, wie der Web Bean Manager entscheidet, welche Implementierung # `ExternalPaymentProcessor` oder `MockPaymentProcessor` # er wählt. Überlegen Sie sich, was passiert, wenn der Manager auf diesen Einspeisungspunkt trifft:

```
@Current PaymentProcessor paymentProcessor
```

Es gibt jetzt zwei Web Beans, die den `PaymentProcessor`-Vertrag erfüllen. Natürlich können wir keine Binding-Annotation zur eindeutig Machung verwenden, da Binding-Annotationen in die Quelle am Einspeisungspunkt hardcodiert und wir wollen, dass der Manager zum Deployment-Zeotpunkt entscheiden können soll!

Die Lösung dieses Problems ist, dass jeder Deployment-Typ eine andere *Präzendenz* besitzt. Die Präzendenz der Deployment-Typen wird durch die Reihenfolge, in der sie in `web-beans.xml` erscheinen, festgelegt. In unserem Beispiel erscheint `@Mock` später als `@Production`, so dass es eine höhere Präzendenz besitzt.

Findet der Manager mehr als ein Web Bean, das den von einem Einspeisungspunkt festgelegten Vertrag erfüllt (API-Typ plus Binding-Annotationen), so gilt die relative Präzedenz der Web Beans. Besitzt eines eine höhere Präzedenz als andere, so wird es für die Einspeisung gewählt. In

unserem Beispiel speist der Web Bean Manager also `MockPaymentProcessor` bei der Ausführung unserer Integrationstestumgebung aus (und das ist es auch, was wir möchten).

Es ist interessant dies mit den heutzutage beliebten Manager Architekturen zu vergleichen. Verschiedene "leichtgewichtige" Container gestatten uns auch das bedingte Deployment von im Klassenpfad existierenden Klassen, aber Klassen, die deployt werden sollen müssen explizit, individuell im Konfigurationscode oder einer XML-Konfigurationsdatei aufgeführt sein. Web Beans unterstützt die Web Bean Definition und Konfiguration via XML, aber im gängigen Fall, in dem keine komplexe Konfiguration erforderlich ist, gestatten Deployment-Types die Aktivierung eines gesamten Satzes von Web Beans mittels einer einzigen XML-Zeile. Währenddessen kann ein den Code durchsehender Entwickler leicht einsehen, in welchen Deployment-Szenarien das Web Bean eingesetzt wird.

4.2.3. Beispiel Deployment-Typen

Deployment-Typen sind hilfreich für allerlei Dinge, hier sind ein paar Beispiele:

- `@Mock` und `@Staging` Deployment-Typen zu Testzwecken
- `@AustralianTaxLaw` für site-spezifische Web Beans
- `@SeamFramework`, `@Guice` für Frameworks Dritter, die auf Web Beans bauen
- `@Standard` für standardmäßige, durch die Web Beans Spezifikation definierte Web Beans

Ihnen fallen sicher noch andere Anwendungen ein...

4.3. Unbefriedigende Abhängigkeiten beheben

Der typensichere Auflösungsalgorithmus schlägt fehl, wenn - nach Betrachtung der Binding-Annotationen und der Deployment-Typen aller den API-Typ implementierender Web Beans eines Einspeisungspunktes - der Web Bean Manager nicht dazu in der Lage ist, ein einzuspeisendes Web Bean zu identifizieren.

Es ist in der Regel einfach, Probleme mit einer `UnsatisfiedDependencyException` oder `AmbiguousDependencyException` zu beheben.

Um eine `UnsatisfiedDependencyException` zu beheben, stellen Sie einfach ein Web Bean bereit, das den API-Typ implementiert und die Binding-Typen des Einspeisungspunkts besitzt# oder aktivieren Sie den Deployment-Typ eines Web Beans, das den API-Typ bereits implementiert und die Binding-Typen besitzt.

Um eine `AmbiguousDependencyException` zu beheben, führen Sie einen Binding-Typ ein, um zwischen den beiden Implementierungen des API-Typs zu unterscheiden oder ändern Sie den Deployment-Typ einer der Implementierungen damit der Web Bean Manager Deployment-Typ Präzedenz zur Auswahl zwischen den beiden verwenden kann. Eine `AmbiguousDependencyException` kann nur vorkommen, wenn zwei Web Beans sich einen Binding-Typ teilen und genau denselben Deployment-Typ besitzen.

Es gibt eine weitere Sache, derer man sich bei der Verwendung von "Dependency"-Einspeisung in Web Beans gewahr sein sollte.

4.4. Client-Proxies

Clients eines eingespeisten Web Beans enthalten in der Regel keinen direkten Verweis an eine Web Bean Instanz.

Stellen wir uns vor, ein an den Geltungsbereich der Anwendung gebundenes Web Bean hielte einen direkten Verweis auf ein an den Geltungsbereich der Anfrage gebundenes Web Bean. Das an den Geltungsbereich der Anwendung gebundene Web Bean wird von vielen verschiedenen Anfragen geteilt. Jedoch sollte jede Anfrage eine andere Instanz des an den Geltungsbereich der Anfrage gebundenen Web Beans sehen!

Stellen Sie sich nun vor das an den Geltungsbereich der Session gebundene Web Bean hielte einen direkten Verweis auf ein an den Geltungsbereich der Anwendung gebundenes Web Bean. FVon Zeit zu Zeit wird der Session Kontext auf Disk serialisiert, um den Speicher effizienter zu nutzen. Die durch den Geltungsbereich der Anwendung begrenzte Instanz des Web Beans sollte jedoch nicht mit dem durch den Geltungsbereich der Session begrenzetn Web Bean serialisiert werden!

Daher muss der Web Bean Manageralle eingespeisten Verweise auf das Web Bean durch ein Proxy-Objekt einleiten, wenn das Web Bean nicht den Standard-Geltungsbereich `@Dependent` besitzt. Dieser *Client-Proxy* ist verantwortlich dafür sicher zu stellen, dass die einen Methodenaufruf erhaltende Web Bean Instanz, die mit dem aktuellen Kontext assoziierte ist. Der Client-Proxy gestattet außerdem die Serialisierung auf Disk von an Kontexte gebundenen Web Beans, ohne dassrekursiv andere eingespeiste Web Beans serialisiert werden.

Leider können aufgrund von Einschränkungen von Java einige Java-Typen nicht vom Web Bean Manager geproxiet werden. Daher meldet der Web Bean Manager eine `UnproxyableDependencyException`, wenn der Typ eines Einspeisungspunkts nicht geproxiet werden kann.

Die folgenden Java-Typen können nicht durch den Web Bean Manager geproxiet werden:

- Als `final` deklarierte Klassen oder die eine `final`-Methode besitzen,
- Klassen, die keinen nicht-privaten Konstruktor ohne Parameter besitzen sowie
- Arrays und primitive Typen.

Es ist in der Regel ganz leicht eine `UnproxyableDependencyException` zu beheben. Fügen Sie der eingespeisten Klasse einfach einen Konstruktor ohne Parameters hinzu, führen Sie ein Interface ein oder ändern Sie den Gelstungsbereich des eingespeisten Web Bean zu `@Dependent`.

4.5. Erhalt eines Web Beans durch programmatsichen "Lookup"

Die Anwendung kann durch Einspeisung eine Instanz des Interface `Manager` erhalten:

```
@Current Manager manager;
```

Das `Manager`-Objekt liefert einen Satz von Methoden zum programmatischen Erhalt einer Web Bean Instanz.

```
PaymentProcessor p = manager.getInstanceByType(PaymentProcessor.class);
```

Binding-Annotationen können durch Subklassifizierung der Helferklasse `AnnotationLiteral` festgelegt werden, da es ansonsten schwierig ist, einen Annotationstyp in Java zu instantiieren.

```
PaymentProcessor p = manager.getInstanceByType(PaymentProcessor.class,
                                    new AnnotationLiteral<CreditCard
>(){});
```

Besitzt der Binding-Typ ein Annotationsmitglied, so können wir keine anonyme Unterklasse von `AnnotationLiteral` # verwenden - stattdessen werden wir eine benannte Unterklasse erstellen müssen:

```
abstract class CreditCardBinding
   extends AnnotationLiteral<CreditCard
>
   implements CreditCard {}
```

```
PaymentProcessor p = manager.getInstanceByType(PaymentProcessor.class,
                                    new CreditCardBinding() {
                                       public void value() { return paymentType; }
                                    } );
```

4.6. Lebenszyklus-Callbacks, @Resource, @EJB und @PersistenceContext

Enterprise Web Beans unterstützen alle durch die EJB-Spezifikation definierten Lebenszyklus-Callbacks: `@PostConstruct`, `@PreDestroy`, `@PrePassivate` und `@PostActivate`.

Einfache Web Beans unterstützen nur die `@PostConstruct` und `@PreDestroy` Callbacks.

Sowohl Enterprise als auch einfache Web Beans unterstützen den Gebrauch von `@Resource`, `@EJB` und `@PersistenceContext` zur Einspeisung von Java EE Ressourcen bzw. EJBs und JPA-Persistenzkontexten. Einfache Web Beans unterstützen den Gebrauch von `@PersistenceContext(type=EXTENDED)` nicht.

Der `@PostConstruct`-Callback erfolgt immer, nachdem alle Abhängigkeiten eingespeist wurden.

4.7. Das `InjectionPoint`-Objekt

Es gibt bestimmte Arten abhängiger Objekte # Web Beans mit Geltungsbereich `@Dependent` # die etwas über das Objekt oder den Einspeisungspunkt in die sie eingespeist werden wissen müssen, um ihre Aufgabe zu erledigen. Zum Beispiel:

- Die Protokollkategorie für einen `Logger` hängt von der Klasse des sie besitzenden Objekts ab.
- Die Einspeisung eines HTTP-Parameters oder Header-Werts hängt davon ab, welcher Parameter oder Header-Name am Einspeisungspunkt festgelegt wurde.
- Einspeisung als Ergebnis der Evaluierung eines EL-Ausdrucks hängt von vom am Einspeisungspunkt festgelegten Ausdruck ab.

Ein Web Bean mit Geltungsbereich `@Dependent` kann eine Instanz von `InjectionPoint` einspeisen und auf Metadaten zugreifen, die mit dem zugehörigen Einspeisungspunkt zu tun haben.

Sehen wir uns ein Beispiel an. Der folgende Code ist umfangreich und empfänglich für Refaktorierungsprobleme:

```
Logger log = Logger.getLogger(MyClass.class.getName());
```

Diese schlaue kleine Producer-Methode gestattet die Einspeisung eines JDK `Logger`, ohne dass explizit eine Protokollkategorie festgelegt werden müsste:

```
class LogFactory {

  @Produces Logger createLogger(InjectionPoint injectionPoint) {
    return Logger.getLogger(injectionPoint.getMember().getDeclaringClass().getName());
```

```
  }

}
```

Wir können jetzt schreiben:

```
@Current Logger log;
```

Sie sind noch nicht ganz überzeugt? Dann sehen Sie sich ein weiteres Beispiel an. Zur Einspeisung von HTTP-Parametern müssen wir einen Binding-Typ festlegen:

```
@BindingType
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER})
public @interface HttpParam {
  @NonBinding public String value();
}
```

Wir würden diesen Binding-Typ an Einspeisungspunkten wie folgt verwenden:

```
@HttpParam("username") String username;
@HttpParam("password") String password;
```

Die folgende Producer-Methode erledigt die Arbeit:

```
class HttpParams

  @Produces @HttpParam("")
  String getParamValue(ServletRequest request, InjectionPoint ip) {
    return request.getParameter(ip.getAnnotation(HttpParam.class).value());
  }

}
```

(Beachten Sie, dass das `value()`-Mitglied der `HttpParam`-Annotation vom Web Bean Manager wird, da es mit `@NonBinding`. annotiert ist)

Der Web Bean Manager liefert ein eingebautes Web Bean, das das `InjectionPoint`-Interface implementiert:

```
public interface InjectionPoint {
   public Object getInstance();
   public Bean<?> getBean();
   public Member getMember():
   public <T extends Annotation
> T getAnnotation(Class<T
> annotation);
   public Set<T extends Annotation
> getAnnotations();
}
```

Geltungsbereiche und Kontexte

Bis jetzt haben wir ein paar Beispiele von *Geltungsbereichtyp-Annotationen* gesehen. Der Geltungsbereich eines Web Beans bestimmt den Lebenszyklus der Instanzen des Web Beans. Der Geltungsbereich bestimmt auch, welche Clients sich auf welche Instanzen des Web Beans beziehen. Gemäß der Web Beans Spezifikation bestimmt ein Geltungsbereich:

- Wann eine neue Instanz eines beliebigen Web Beans mit diesem Geltungsbereich erstellt wird
- Wenn eine bestehende Instanz eines beliebigen Web Beans mit diesem Geltungsbereich gelöscht wird
- Welche eingespeisten Referenzen auf eine beliebige Instanz eines Web Beans mit diesem Geltungsbereich verweisen

Wenn wir etwa ein session-begrenztes Web Bean `CurrentUser` haben, so sehen alle Web Beans, die im Kontext derselben `HttpSession` aufgerufen werden, dieselbe Instanz von `CurrentUser`. Diese Instanz wird automatisch erstellt, wenn `CurrentUser` erstmals in dieser Session benötigt wird und automatisch gelöscht, wenn die Session endet.

5.1. Typen von Geltungsbereichen

Web Beans besitzen ein *erweiterbares Kontextmodell*. Es ist möglich, neue Geltungsbereiche zu definieren, indem man eine neue Annotation für einen Geltungsbereich-Typ erstellt:

```
@Retention(RUNTIME)
@Target({TYPE, METHOD})
@ScopeType
public @interface ClusterScoped {}
```

Natürlich ist dies der einfache Teil des Jobs. Damit dieser Typ von Geltungsbereich von Nutzen ist, müssen wir außerdem ein `Context`-Objekt definieren, das den Geltungsbereich implementiert! Die Implementierung eines `Context` ist in der Regel ein sehr technisches Unterfangen, das nur für Framework-Entwicklung vorgesehen ist.

Wir können eine Annotation eines Geltungsbereich-Typs an einer Web Bean Implementierungsklasse anwenden, um den Geltungsbereich des Web Beans festzulegen:

```
@ClusterScoped
public class SecondLevelCache { ... }
```

In der Regel verwenden Sie einen der eingebauten Geltungsbereiche der Web Beans.

5.2. Eingebaute Geltungsbereiche

Web Beans definiert vier eingebaute Geltungsbereiche:

- `@RequestScoped`
- `@SessionScoped`
- `@ApplicationScoped`
- `@ConversationScoped`

Für eine Web Beans verwendende Webanwendung:

- Jede Servlet-Anfrage hat Zugriff auf die aktuelle Anfrage, Session und Geltungsbereiche der Anwendung und zusätzlich
- hat jede JSF-Anfrage Zugriff auf einen aktiven Konversations-Geltungsbereich.

Die Geltungsbereiche von Anfrage und Applikation sind ebenfalls aktiv:

- während Aufrufen von EJB Remote-Methoden,
- während EJB-Timeouts,
- während Message Delivery an ein message-betriebenes Bean und
- während Aufrufen von Webdiensten.

Versucht die Applikation ein Web Bean aufzurufen, das keinen aktiven Kontext besitzt, so wird zur Runtime eine `ContextNotActiveException` vom Web Bean Manager gemeldet.

Drei der vier eingebauten Geltungsbereiche solltem jedem Java EE Entwickler sehr vertraut sein, daher wollen wir hier nicht weiter auf diese eingehen. Einer der Geltungsbereiche jedoch ist neu.

5.3. Der Geltungsbereich der Konversation

Der Web Beans Geltungsbereich der Konversation ähnelt dem herkömmlichen Geltungsbereich der Session dahingehend, dass er den mit einem Benutzer des Systems assoziierten Status verwahrt und mehrere Anfragen zum Server umfasst. Anders als für den Geltungsbereich der Session gilt für den Geltungsbereich der Konversation jedoch:

- ist explizit durch die Applikation demarkiert und
- verwahrt den mit einem bestimmten Webbrowser assoziierten Status in einer JSF-Applikation.

Eine Konversation repräsentiert aus Perspektive des Benutzers eine Aufgabe, eine Arbeitseinheit. Der Konversationskontext enthält den Status dessen, woran der Benutzer gerade arbeitet. Arbeitet der Benutzer gleichzeitig an mehreren Dingen, so existieren mehrere Konversationen.

Der Konversation skontext ist während jeder JSF-Anfrage aktiv. Jedoch werden die meisten Konversationen am Ende der Anfrage gelöscht. Soll eine Konversation den Status über mehrere Anfragen hinweg verwahren, so muss sie explizit zu einer *lange laufenden Konversation* fortgepflanzt werden.

5.3.1. Konversationsdemarkierung

Web Beans liefert ein eingebautes Web Bean für die Steuerung des Lebenszyklus von Konversationen in einer JSF-Applikation. Dieses Web Bean kann durch Einspeisung erhalten werden:

```
@Current Conversation Konversation;
```

Um die mit der aktuellen Anfrage assoziierte Konversation an eine lange laufende Konversation fortzupflanzen, rufen Sie die `begin()`-Methode vom Applikationscode auf. Um den aktuellen, lange laufenden Konversationskontext für die Löschung am Ende der aktuellen Anfrage zu terminieren, rufen Sie `end()` auf.

IIm folgenden Beispiel steuert ein konversationsbegrenztes Web Bean die Konversation, mit der es assoziiert ist:

```
@ConversationScoped @Stateful
public class OrderBuilder {

   private Order order;
   private @Current Conversation conversation;
   private @PersistenceContext(type=EXTENDED) EntityManager em;

   @Produces public Order getOrder() {
      return order;
   }

   public Order createOrder() {
      order = new Order();
      conversation.begin();
      return order;
   }

   public void addLineItem(Product product, int quantity) {
      order.add( new LineItem(product, quantity) );
```

```
    }

    public void saveOrder(Order order) {
        em.persist(order);
        conversation.end();
    }

    @Remove
    public void destroy() {}

}
```

Dieses Web Bean ist in der Lage, seinen eigenen Lebenszyklus durch Verwendung der `Conversation`-API zu steuern. Aber einige andere Web Beans besitzen einen Lebenszyklus, der komplett von einem anderen Objekt abhängt.

5.3.2. Konversationsfortpflanzung (Conversation Propagation)

Der Konversationskontext wird automatisch mit allen JSF Faces Anfragen fortgepflanzt(JSF-Formulareinreichung). Nicht-Faces Anfragen werden nicht automatisch fortgepflanzt, zum Beispiel Navigation via einem Link.

Wir können die Konversation zwingen, mit einer nicht-Faces Anfrage fortzupflanzen, indem wir den eindeutigen Bezeichner der Konversation als einen Anfragenparameter mit einschließen. Die Web Beans Spezifikation reserviert den Anfragenparameter namens `cid` für diesen Gebrauch. Den eindeutigen Bezeichner der Konversation erhält man vom `Conversation`-Objekt, welches den Web Beans Namen `conversation` besitzt.

Daher pflanzt das folgende Link die Konversation fort:

```
<a href="/addProduct.jsp?cid=#{conversation.id}"
>Add Product</a
>
```

Der Web Bean Manager muss auch Konversationen über ein Redirect fortpflanzen, selbst wenn die Konversation nicht als lange laufend gekennzeichnet ist. Dies macht die Implementierung des POST-then-redirect Musters sehr einfach, ohne dass man sich auf anfällige Konstrukte wie etwa ein "Flash"-Objekt stützen müsste. In tdiesem Fall fügt der Web Bean Manager automatisch einen Anfragenparameter hinzu, um die URL umzuleiten.

5.3.3. Konversations-Timeout

Dem Web Bean Manager ist gestattet, eine Konversation und alle Stati innerhalb seines Kontexts zu jedem Zeitpunkt zu löschen, um Ressourcen zu schonen. Eine Implementierung des Web Bean

Managers wird dies in der Regel auf der Basis einer Art von Timeout # tun, obwohl dies nicht durch die Web Beans Spezifikation gefordert wird. Beim Timeout handelt es sich um den Zeitraum von Inaktivität, ehe die Konversation gelöscht wird.

Das `Conversation`-Objekt liefert eine Methode, mit der der Timeout eingestellt werden kann. Dies ist ein Tipp an den Web Bean Manager, der die Einstellung ignorieren kann.

```
conversation.setTimeout(timeoutInMillis);
```

5.4. Der abhängige Pseudo-Geltungsbereich ("Pseudo-Scope")

Neben den vier eingebauten Geltungsbereichen bieten Web Beans den sogenannten*abhängigen Pseudo-Geltungsbereich*. Dies ist der standardmäßige Geltungsbereich für ein Web Bean, das nicht expliziet einen Typ von Geltungsbereich deklariert.

Zum Beispiel besitzt dieses Web Bean den Geltungsbereich-Typ `@Dependent`:

```
public class Calculator { ... }
```

Wenn der Einspeisungspunkt eines Web Bean zu einem abhängigen Web Bean hin aufgelöst wird, so wird jedes Mal, wenn das erste Web Bean instantiiert wird, eine neue Instanz des abhängigen Web Beans erstellt. Instanzen abhängiger Web Beans werden nie von unterschiedlichen Web Beans oder unterschiedlichen Einspeisungspunkten geteilt. Sie sind *abhängige Objekte* einer anderen Web Bean Instanz.

Abhängige Web Bean Instanzen werden gelöscht, wenn die Instanz von der sie abhängen gelöscht wird.

Web Beans machen es einfach, eine unabhängige Instanz einer Java-Klasse oder eines EJB-Beans zu erhalten, selbst wenn die Klasse oder das EJB-Bean bereits als ein Web Bean mit einem anderen Typ von Geltungsbereich deklariert sind.

5.4.1. Die `@New`-Annotation

Die eingebaute `@New` Binding-Annotation gestattet die *implizite* Definition eines abhängigen Web Beans an einem Einspeisungspunkt. Nehmen wir an, wir deklarieren das folgende eingespeiste Feld:

```
@New Calculator calculator;
```

Dann wird ein Web Bean mit Geltungsbereich `@Dependent`, Binding-Typ `@New`, API-Typ `Calculator`, Implementierungsklasse `Calculator` und Deployment-Typ `@Standard` impliziert definiert.

Dies ist wahr, selbst wenn `Calculator` *bereits* mit einem anderen Typ von Geltungsbereich definiert ist, zum Beispiel:

```
@ConversationScoped
public class Calculator { ... }
```

Die folgenden eingespeisten Attribute erhalten also jeweils eine unterschiedliche Instanz von `Calculator`:

```
public class PaymentCalc {

   @Current Calculator calculator;
   @New Calculator newCalculator;

}
```

In das `calculator`-Feld wird eine konversationsbegrenzte Instanz von `Calculator` eingespeist. In das `newCalculator`-Feld wird eine neue Instanz von `Calculator` eingespeist, mit einem Lebenszyklus, der an den besitzenden `PaymentCalc` gebunden ist.

Dieses Feature ist insbesondere im Zusammenhang mit Producer-Methoden von Nutzen, wie wir im folgenden Kapitel noch sehen werden.

Producer-Methoden

Producer-Methoden gestatten es uns, bestimmte Beschränkungen zu umgehen, die auftreten, wenn der Web Bean Manager statt die Anwendung für die Instantiierung von Objekten verantwortlich ist. Sie sind auch die einfachste Art der Integration von Objekten in die Web Beans Umgebung, die keine Web Beans sind. (In *Kapitel 12, Definition von Web Beans unter Verwendung von XML* lernen wir eine zweite Weise kennen.)

Gemäß der Spezifikation:

> A Web Beans producer method acts as a source of objects to be injected, where:
>
> - the objects to be injected are not required to be instances of Web Beans,
> - the concrete type of the objects to be injected may vary at runtime or
> - the objects require some custom initialization that is not performed by the Web Bean constructor

For example, producer methods let us:

- expose a JPA entity as a Web Bean,
- expose any JDK class as a Web Bean,
- define multiple Web Beans, with different scopes or initialization, for the same implementation class, or
- vary the implementation of an API type at runtime.

In particular, producer methods let us use runtime polymorphism with Web Beans. As we've seen, deployment types are a powerful solution to the problem of deployment-time polymorphism. But once the system is deployed, the Web Bean implementation is fixed. A producer method has no such limitation:

```
@SessionScoped
public class Preferences {

   private PaymentStrategyType paymentStrategy;

   ...

   @Produces @Preferred
   public PaymentStrategy getPaymentStrategy() {
      switch (paymentStrategy) {
```

```
            case CREDIT_CARD: return new CreditCardPaymentStrategy();
            case CHEQUE: return new ChequePaymentStrategy();
            case PAYPAL: return new PayPalPaymentStrategy();
            default: return null;
        }
    }

}
```

Consider an injection point:

```
@Preferred PaymentStrategy paymentStrat;
```

This injection point has the same type and binding annotations as the producer method, so it resolves to the producer method using the usual Web Beans injection rules. The producer method will be called by the Web Bean manager to obtain an instance to service this injection point.

6.1. Geltungsbereich einer Producer-Methode

Der Geltungsbereich der Producer-Methode ist standardmäßig `@Dependent`, und daher wird sie *jedes Mal* aufgerufen, wenn der Web Bean Manager eine Einspeisung in dieses oder ein anderes in diese Producer-Methode auflösendes Feld vornimmt. Es könnten daher mehrere Instanzen des `PaymentStrategy`-Objekts für jede Benutzer-Session vorhanden sein.

Um dieses Verhalten zu ändern, können wir der Methode eine `@SessionScoped`-Annotation hinzufügen.

```
@Produces @Preferred @SessionScoped
public PaymentStrategy getPaymentStrategy() {
    ...
}
```

Wird jetzt die Producer-Methode aufgerufen, so wird die wiedergegebene `PaymentStrategy` an den Session-Kontext gebunden. Die Producer-Methode wird in drselben Session nicht mehr aufgerufen.

6.2. Einspeisung in Producer-Methoden

Es gibt ein potenzielles Problem mit dem Code oben. Die Implementierungen von `CreditCardPaymentStrategy` werden unter Verwendung des Java `new` Operators instantiiert. Direkt durch die Anwendung instantiierte Objekte können die Dependency-Einspeisung nicht nutzen und besitzen keine Interzeptoren.

Falls dies nicht das ist was wir wünschen, so können wir Dependency-Einspeisung in die Producer-Methode verwenden, um Web Bean Instanzen zu erhalten:

```
@Produces @Preferred @SessionScoped
public PaymentStrategy getPaymentStrategy(CreditCardPaymentStrategy ccps,
                                          ChequePaymentStrategy cps,
                                          PayPalPaymentStrategy ppps) {
   switch (paymentStrategy) {
      case CREDIT_CARD: return ccps;
      case CHEQUE: return cps;
      case PAYPAL: return ppps;
      default: return null;
   }
}
```

Aber Moment mal, was wenn `CreditCardPaymentStrategy` ein anfragenbegrenztes Web Bean ist? Dann hat die Producer-Methode die Wirkung, dass Sie die aktuelle anfragenbegrenzte Instanz in den Geltungsbereich der Session "befördert". Das ist mit ziemlicher Sicherheit ein Fehler! Das anfragenbegrenzte Objekt wird vom Web Bean Manager gelöscht ehe die Session endet, aber der Verweis auf das Objekt bleibt im Geltungsbereich der Session "hängen" in the session scope. Dieser Fehler wird *nicht* vom Web Bean Manager aufgespürt, daher seien Sie besonders vorsichtig wenn Sie Web Bean Instanzen von Producer-Methoden wiedergeben!

Es existieren mindestens drei Arten, wie dieser Fehler behoben werden kann. Wir könnten den Geltungsbereich der `CreditCardPaymentStrategy`-Implementierung ändern, aber das würde auch andere Clients dieses Web Beans betreffen. Eine bessere Option wäre es, den Geltungsbereich der Producer-Methode auf `@Dependent` oder `@RequestScoped` zu ändern.

Eine gängigere Lösung ist es jedoch, die spezielle `@New` Binding-Annotation zu verwenden.

6.3. Verwendung von `@New` mit Producer-Methoden

Sehen Sie sich folgende Producer-Methode an:

```
@Produces @Preferred @SessionScoped
public PaymentStrategy getPaymentStrategy(@New CreditCardPaymentStrategy ccps,
                                          @New ChequePaymentStrategy cps,
                                          @New PayPalPaymentStrategy ppps) {
   switch (paymentStrategy) {
      case CREDIT_CARD: return ccps;
      case CHEQUE: return cps;
      case PAYPAL: return ppps;
      default: return null;
   }
```

```
}
```

Dann wird eine neue *abhängige* Instanz von `CreditCardPaymentStrategy` erstellt, an die Producer-Methode weitergegeben, von der Producer-Methode wiedergegeben und schließlich an den Session-Kontext gebunden. Das abhängige Objekt wird nicht gelöscht bis das `Preferences`-Objekt gelöscht wird, meist am Ende der Session.

Teil II. Entwicklung lose gepaarten Codes

Das erste wichtige Thema von Web Beans ist *Loose Coupling* (lose Paarung). Wir haben bereits drei Wege gesehen, diese lose Paarung zu erreichen:

- *Deployment-Typen* aktivieren Polymorphie zum Zeitpunkt des Deployment,
- *Producer Methoden* aktivieren Polymorphie zur Runtime, und
- *kontextuelles Lebenszyklus-Management* entkoppelt Web Bean Lebenszyklen.

Diese Techniken dienen der Aktivierung der losen Paarung ("Loose Coupling") von Client und Server. Der Client ist nicht mehr eng an eine API-Implementierung gebunden und muss den Lebenszyklus des Server-Objekts nicht mehr verwalten. Dadurch können *Objekte, die "stateful" sind, interagieren als seien Sie Dienste* .

Lose Paarung macht ein System *dynamischer*. Das System kann auf gut definierte Weise auf Änderungen reagieren. In der Vergangenheit war es der Fall, dass Frameworks die versuchten die obigen Facilities bereitzustellen, dies auf Kosten der Typensicherheit taten. Bei Web Beans handelt es sich um die erste Technologie, die diese Ebene der losen Paarung auf typensichere Weise ermöglicht.

Web Beans bieten drei weitere wichtige Facilities, die das Ziel loser Paarung weiterbringen:

- *Interzeptoren* entkoppeln technische Probleme von Business Logik,
- *Dekoratoren* ("Decorators") können eingesetzt werden, um einige Business Probleme zu entkoppeln und
- *Ereignis Benachrichtigungen* ("Event Notifications") entkoppeln Ereignis-Producer von Ereignis-Konsument.

Sehen wir uns zunächst die Interzeptoren an.

Interzeptoren

Web Beans verwenden die grundlegende Interzeptor-Architektur von EJB 3.0, wobei die Funktionalität in zwei Richtungen erweitert wird:

- Jedes Web Bean kann Interzeptoren besitzen, nicht nur Session Beans.
- Web Beans bieten eine fortgeschrittenere auf Annotationen basierende Vorgehensweise bei der Bindung von Interzeptoren an Web Beans.

Die EJB-Spezifikation definiert zwei Arten von Abfangpunkten (sog. "Interception Points"):

- Business Methoden Interception und
- Lebenszyklus-Callback Interception.

Ein *Business Methoden Interzeptor* gilt für Aufrufe von Methoden des Web Beans durch Clients des Web Beans:

```
public class TransactionInterceptor {
   @AroundInvoke public Object manageTransaction(InvocationContext ctx) { ... }
}
```

Ein *Lebenszyklus Callback-Interzeptor* gilt für Aufrufe von Lebenszyklus Callbacks durch den Container:

```
public class DependencyInjectionInterceptor {
   @PostConstruct public void injectDependencies(InvocationContext ctx) { ... }
}
```

Eine Interzeptorklasse kann sowohl Lebenszyklus-Callbacks als auch Business-Methoden abfangen.

7.1. Interzeptor-Bindings

Nehmen wir an, wir wollten deklarieren, dass einige unserer Web Beans transaktional sind. Das erste, was wir benötigen ist eine *Interzeptor bindende Annotation*, um festzulegen, für welches Web Bean wir uns interessieren:

```
@InterceptorBindingType
@Target({METHOD, TYPE})
```

```
@Retention(RUNTIME)
public @interface Transactional {}
```

Jetzt können wir ganz leicht unser `ShoppingCart` als ein transaktionales Objekt festlegen:

```
@Transactional
public class ShoppingCart { ... }
```

Oder, falls uns das lieber ist, können wir festlegen, dass nur eine Methode transaktional ist:

```
public class ShoppingCart {
   @Transactional public void checkout() { ... }
}
```

7.2. Implementierung von Interzeptoren

Das ist toll, aber irgendwann müssen wir den den Managementaspekt dieser Transaktion liefert, implementieren. Wir müssen nur einen standardmäßigen EJB-Interzeptor erstellen und ihn mit `@Interceptor` und `@Transactional` annotieren.

```
@Transactional @Interceptor
public class TransactionInterceptor {
   @AroundInvoke public Object manageTransaction(InvocationContext ctx) { ... }
}
```

Bei allen Web Beans Interzeptoren handelt es sich um einfache Web Beans und sie können "Dependency"-Einspeisung und kontextuelles Lebenszyklus-Management nutzen.

```
@ApplicationScoped @Transactional @Interceptor
public class TransactionInterceptor {

   @Resource Transaction transaction;

   @AroundInvoke public Object manageTransaction(InvocationContext ctx) { ... }

}
```

Mehrere Interzeptoren können denselben Interzeptor Binding-Typ verwenden.

7.3. Interzeptoren aktivieren

Schließlich müssen wir unseren Interzeptor in `web-beans.xml` *aktivieren*.

```
<Interceptors>
   <tx:TransactionInterceptor/>
</Interceptors
>
```

Puh! Warum diese Suppe an Klammern?

Nun, die XML-Deklaration löst zwei Probleme:

- Sie ermöglicht es uns, eine totale Reihenfolge für alle Interzeptoren in unserem System festzulegen, wodurch deterministisches Verhalten festgelegt wird und
- Interzeptor-Klassen zum Zeitpunkt des Deployments aktiviert oder deaktiviert werden können.

Zum Beispiel könnten wir festlegen, dass unser Sicherheitsinterzeptor vor unserem `TransactionInterceptor` ausgeführt wird.

```
<Interceptors>
   <sx:SecurityInterceptor/>
   <tx:TransactionInterceptor/>
</Interceptors
>
```

Oder wir könnten sie beide in unserer Testumgebung abschalten!

7.4. Interzeptor-Bindings mit Mitgliedern

Nehmen wir an, wir wollten unserer `@Transactional`-Annotation weitere Informationen hinzufügen:

```
@InterceptorBindingType
@Target({METHOD, TYPE})
@Retention(RUNTIME)
public @interface Transactional {
   boolean requiresNew() default false;
}
```

Web Beans verwendet den Wert von `requiresNew` zur Auswahl zwischen zwei verschiedenen Interzeptoren `TransactionInterceptor` und `RequiresNewTransactionInterceptor` auszuwählen.

```
@Transactional(requiresNew=true) @Interceptor
public class RequiresNewTransactionInterceptor {
   @AroundInvoke public Object manageTransaction(InvocationContext ctx) { ... }
}
```

Jetzt können wir `RequiresNewTransactionInterceptor` wie folgt verwenden:

```
@Transactional(requiresNew=true)
public class ShoppingCart { ... }
```

Was aber, wenn wir über nur einen Interzeptor verfügen und wir wollen, dass der Manager bei der Bindung der Interzeptoren den Wert von `requiresNew` ignoriert? Wir können die `@NonBinding`-Annotation verwenden:

```
@InterceptorBindingType
@Target({METHOD, TYPE})
@Retention(RUNTIME)
public @interface Secure {
   @NonBinding String[] rolesAllowed() default {};
}
```

7.5. Multiple Interzeptor bindende Annotationen

In der Regel verwenden wir Kombinationen von Interzeptor-Binding-Typen, um mehrere Interzeptoren an ein Web Bean zu binden. Folgende Deklaration etwa würde verwendet, um `TransactionInterceptor` und `SecurityInterceptor` an dasselbe Web Bean zu binden:

```
@Secure(rolesAllowed="admin") @Transactional
public class ShoppingCart { ... }
```

In sehr komplexen Fällen aber kann ein Interzeptor selbst eine Kombination von Interzeptor-Binding-Typen festlegen:

```
@Transactional @Secure @Interceptor
```

```
public class TransactionalSecureInterceptor { ... }
```

Dann könnte dieser Interzeptor an die `checkout()`-Methode gebunden werden, indem eine der folgenden Kombinationen verwendet wird:

```
public class ShoppingCart {
   @Transactional @Secure public void checkout() { ... }
}
```

```
@Secure
public class ShoppingCart {
   @Transactional public void checkout() { ... }
}
```

```
@Transactionl
public class ShoppingCart {
   @Secure public void checkout() { ... }
}
```

```
@Transactional @Secure
public class ShoppingCart {
   public void checkout() { ... }
}
```

7.6. Vererbung von Interzeptor-Binding-Typen

Eine Einschränkung des Java Sprach-Supports für Annotationen ist das Fehlen von Annotationsvererbung. Eigentlich sollten Annotationen eine eingebaute Wiederverwendung besitzen, damit diese Art von Sache funktioniert:

```
public @interface Action extends Transactional, Secure { ... }
```

Nun, zum Glück umgeht Web Beans dieses fehlende Feature von Java. Wir können einen Interzeptor Binding-Typ mit anderen Interzeptor Binding-Typen annotieren. Die Interzeptor-Bindings sind transitive # jedes Web Bean mit demselben Interzeptor-Binding erbt die als Meta-Annotationen deklarierten Interzeptor-Bindings.

```
@Transactional @Secure
@InterceptorBindingType
@Target(TYPE)
@Retention(RUNTIME)
public @interface Action { ... }
```

Jedes mit `@Action` annotierte Web Bean wird sowohl an `TransactionInterceptor` als auch `SecurityInterceptor` gebunden. (Und sogar an `TransactionalSecureInterceptor`, falls es existiert).

7.7. Verwendung von `@Interceptors`

Die durch die EJB-Spezifikation definierte `@Interceptors`-Annotation wird sowohl für Enterprise als auch einfache Web Beans unterstützt, zum Beispiel:

```
@Interceptors({TransactionInterceptor.class, SecurityInterceptor.class})
public class ShoppingCart {
   public void checkout() { ... }
}
```

Allerdings besitzt diese Vorgehensweise folgende Nachteile:

- Die Interzeptorimplementierung ist im Business Code hardkodiert,
- Interzeptoren können zum Deployment-Zeitpunkt nicht einfach deaktiviert werden und
- Die Interzeptorreihenfolge ist nicht allgemeingültig # sie wird durch die Reihenfolge, in der Interzeptoren auf Klassenebene aufgeführt sind, festgelegt.

Daher empfehlen wir die Verwendung von Interzeptor-Bindings im Web Beans Stil.

Dekoratoren

Interzeptoren bieten eine leistungsfähige Weise, Probleme, die *orthogonal* zum Typensystem sind, festzuhalten und zu trennen. Jeder Interzeptor kann Aufrufe jedes Java Typs abfangen. Dies macht ihn perfekt für die Lösung technischer Probleme wie etwa Transaktionsmanagement und Sicherheit.Jedoch sind Interzeptoren ihrem Wesen nach nicht der tatsächlichen Semantik der Ereignisse gewahr, die sie abfangen. Interzeptoren sind daher nicht die geeigneten Tools zur Separierung von unternehmensbezogenen Problemen.

Das Gegenteil gilt für *Dekoratoren*. Ein Dekorator fängt Aufrufe nur für ein bestimmtes Java-Interface ab und kennt daher die zu diesem Interface gehörende Semantik. Dadurch sind Dekoratoren das perfekte Tool zur Bearbeitung einige unternehmensbezogener Probleme. Es bedeutet auch, dass Dekoratoren nicht diesselbe Allgemeingültigkeit wie Interzeptoren besitzen. Dekoratoren können keine technischen Probleme lösen, die sich über viele disparate Typen verteilen.

Nehmen wir an, wir besitzen ein Konten repräsentierendes Interface:

```
public interface Account {
   public BigDecimal getBalance();
   public User getOwner();
   public void withdraw(BigDecimal amount);
   public void deposit(BigDecimal amount);
}
```

Mehrere verschiedene Web Beans in unserem System implementieren das `Account`-Interface. Es existiert allerdings eine gängige legale Voraussetzung die besagt, dass für jede Art von Konto, große Transaktionen vom System in einem besonderen Protokoll gespeichert werden müssen. Dies ist die perfekte Aufgabe für einen Dekorator.

Ein Dekorator ist ein einfaches Web Bean, das den Typ das es dekoriert implementiert und `@Decorator` annotiert ist.

```
@Decorator
public abstract class LargeTransactionDecorator
      implements Account {

   @Decorates Account account;

   @PersistenceContext EntityManager em;

   public void withdraw(BigDecimal amount) {
      account.withdraw(amount);
```

```
        if ( amount.compareTo(LARGE_AMOUNT)
>0 ) {
            em.persist( new LoggedWithdrawl(amount) );
        }
    }

    public void deposit(BigDecimal amount);
        account.deposit(amount);
        if ( amount.compareTo(LARGE_AMOUNT)
>0 ) {
            em.persist( new LoggedDeposit(amount) );
        }
    }

}
```

Anders als andere einfache Web Beans, kann ein Dekorator eine abstrakte Klasse sein. Falls es nichts besonderes ist, dass der Dekorator für eine bestimmte Methode des dekorierten Interface tun muss, so brauchen Sie diese Methode nicht zu implementieren.

8.1. "Delegate" Attribute

Alle Dekoratoren besitzen ein *"Delegate" Attribut*. Typ und Binding-Typen des "Delegate" Attribut bestimmen, an welche Web Beans der Dekorator gebunden wird. Der Typ des "Delegate" Attributs muss alle vom Dekorator implementierten Interfaces implementieren oder erweitern.

Dieses "Delegate" Attribut legt fest, dass der Dekorator an alle `Account` implementierenden Web Beans gebunden wird:

```
@Decorates Account account;
```

Ein "Delegate" Attribut kann eine Binding-Annotation festlegen. Dann wird der Dekorator nur an Web Beans mit demselben Binding gebunden.

```
@Decorates @Foreign Account account;
```

Ein Dekorator wird an ein beliebiges Web Bean gebunden, das:

- den Typ des "Delegate" Attributs als einen API-Typ hat und
- alle Binding-Typen besitzt die durch das "Delegate" Attribut deklariert werden.

Der Dekorator kann das "Delegate" Attribut aufrufen, was eine sehr ähnliche Wirkung wie der Aufruf von `InvocationContext.proceed()` von einem Interzeptor hat.

8.2. Aktivierung von Dekoratoren

Wir müssen unseren Dekorator in `web-beans.xml` *aktivieren*.

```
<Decorators>
   <myapp:LargeTransactionDecorator/>
</Decorators
>
```

Diese Deklaration dient demselben Zweck für Dekoratoren, den die`<Interceptors>`-Deklaration für Interzeptoren erfüllt:

- es ermöglicht uns eine gesamte Ordnung für alle Dekoratoren in unserem System festzulegen, wodurch deterministisches Verhalten gewährleistet wird und
- es gestattet uns, Dekorator-Klassen zum Deplyment-Zeitpunkt zu aktivieren oder zu deaktivieren.

Interzeptoren für eine Methode werden aufgerufen vor den Dekoratoren an dieser Methode angewendet werden.

Ereignisse

Die Web Beans Ereignisbenachrichtigungs-Facility gestattet es Web Beans auf eine völlig abgekoppelte Weise zu interagieren. Ereignis *Producer* bringen Ereignisse auf, die dann vom Web Bean Manager an Ereignis *Observer* geliefert werden. Dieses schlichte Schema klingt zwar etwas nach dem bekannten Observer/observierbar Muster, aber es gibt ein paar Überraschungen:

- nicht nur sind Ereignis-Producer von Observern abgekoppelt, Observer sind auch komplett von Producern abgekoppelt,
- Observer können eine Kombination von "Selektoren" festlegen, um den Satz von Ereignisbenachrichtigungen einzugrenzen, die sie erhalten und
- Observer können sofort benachrichtigt werden oder sie können festlegen, dass die Lieferung des Ereignisses bis zum Abschluss der aktuellen Transaktion verschoben wird

9.1. Ereignis-Observer

Eine *Observer-Methode* ist eine Methode eines Web Beans mit einem Parameter, der `@Observes` annotiert ist.

```
public void onAnyDocumentEvent(@Observes Document document) { ... }
```

Der annotierte Parameter heißt *Ereignisparameter*. Der Typ des Ereignisparameter ist der beobachtete *Ereignistyp*. Observer-Methoden können auch "Selektoren" festlegen, die nur Instanzen von von Web Beans Binding-Typen sind. Wird ein Binding-Typ als Ereignis-Selektor verwendet, so wird dies als *Ereignis Binding-Typ*.

```
@BindingType
@Target({PARAMETER, FIELD})
@Retention(RUNTIME)
public @interface Updated { ... }
```

Wir legen die Ereignis-Bindings der Observer-Methode durch Annotation des Ereignisparameters fest:

```
public void afterDocumentUpdate(@Observes @Updated Document document) { ... }
```

Eine Observer-Methode muss keine Ereignis-Bindings # festlegen, in diesem Fall interessiert sie sich für *alle* Ereignisse eines bestimmten Typs. Legt sie Ereignis-Bindings fest, so interessiert sie sich nur für Ereignisse, die diese Ereignis-Bindings besitzen.

Die Observer-Methode kann zusätzliche Parameter besitzen, die gemäß der üblichen Einspeisungssemantik Web Beans Methodenparameter eingespeist werden:

```
public void afterDocumentUpdate(@Observes @Updated Document document, User user) { ... }
```

9.2. Ereignis-Producer

Der Ereignis-Producer kann ein *Ereignisbenachrichtigungs*-Objekt durch Einspeisung erhalten:

```
@Observable Event<Document
> documentEvent
```

Die `@Observable`-Annotation definiert implizit ein Web Bean mit Geltungsbereich `@Dependent` und Deployment-Typ `@Standard` mit einer durch den Web Bean Manager bereitgestellten Implementierung.

Ein Producer bringt durch Aufruf der `fire()`-Methode des `Event`-Interface Ereignisse auf, wobei ein *Ereignisobjekt* weitergegeben wird:

```
documentEvent.fire(document);
```

Ein Ereignis-Objekt kann eine Instanz einer beliebigen Java-Klasse sein, die keine Typ-Variablen oder Platzhalter-Typenparameter besitzt. Das Ereignis wird an jede Observer-Methode geliefert, die:

- einen Ereignisparameter besitzt, dem das Ereignisobjekt zugeschrieben werden kann und
- keine Ereignis-Bindings festlegt.

Der Web Bean Manager ruft einfach alle Observer-Methoden auf und gibt das Ereignis-Objekt als den Wert des Ereignisparameters weiter. Meldet eine Observer-Methode eine Ausnahme, so stoppt der Web Bean Manager den Aufruf von Observer-Methoden und die Ausnahme wird durch die `fire()`-Methode erneut gemeldet.

Um einen "Selektor" festzulegen kann der Ereignis-Producer eine Instanz des Ereignis-Binding-Typs an die `fire()`-Methode weitergeben:

```
documentEvent.fire( document, new AnnotationLiteral<Updated
>(){} );
```

Die Helferklasse `AnnotationLiteral` ermöglicht die Instanziierung der Binding-Typen inline, da es andernfalls schwierig ist, die in Java zu tun.

Das Ereignis wird an jede Observer-Methode geliefert, die:

- einen Ereignisparameter besitzt, dem das Ereignisobjekt zugeschrieben werden kann und
- kein Ereignis-Binding festlegt *außer* für die an `fire()` weitergegebenen Ereignis-Bindings.

Alternativ können Ereignis-Bindings durch Festlegen des Einspeisungspunkts der Ereignisbenachrichtigung festgelegt werden:

```
@Observable @Updated Event<Document
> documentUpdatedEvent
```

Dann besitzt jedes über diese Instanz abgegebene Ereignis das `Event` annotierte Ereignis-Binding. Das Ereignis wird an jede Observer-Methode geliefert, die:

- einen Ereignisparameter besitzt, dem das Ereignisobjekt zugeschrieben werden kann und
- Kein Ereignis-Binding festlegt *außer* für die an `fire()` oder die annotierten Ereignis-Bindings des Einspeisungspunkts für Ereignisbenachrichtigungen weitergegebenen.

9.3. Dynamische Registrierung von Observern

Es ist oft hilfreich, einen Ereignis-Observer dynamisch zu registrieren. Die Anwendung kann das `Observer`-Interface implementieren und eine Instanz mit einer Ereignisbenachrichtigung registrieren, indem die `observe()`-Methode aufgerufen wird.

```
documentEvent.observe( new Observer<Document
>() { public void notify(Document doc) { ... } } );
```

Typen von Ereignis-Bindings können durch den Einspeisungspunkt für Ereignisbenachrichtigungen oder Weitergabe von Instanzen von Typen von Ereignis-Bindings an die `observe()`-Methode festgelegt werden:

```
documentEvent.observe( new Observer<Document
>() { public void notify(Document doc) { ... } },
                                    new AnnotationLiteral<Updated
>(){} );
```

9.4. Ereignis-Bindings mit Mitgliedern

Ein Ereignis-Binding-Typ kann Annotationsmitglieder besitzen:

```
@BindingType
@Target({PARAMETER, FIELD})
@Retention(RUNTIME)
public @interface Role {
   RoleType value();
}
```

Der Mitgliederwert dient der Eingrenzung von an den Observer gelieferten Nachrichten:

```
public void adminLoggedIn(@Observes @Role(ADMIN) LoggedIn event) { ... }
```

Typenmitglieder von Ereignis-Bindings können durch den Ereignis-Producer statisch festgelegt werden - dies erfolgt über Annotationen am Einspeisungspunkt der Ereignisbenachrichtigungen:

```
@Observable @Role(ADMIN) Event<LoggedIn
> LoggedInEvent;}}
```

Alternativ kann der Wert des Typenmitglieds des Ereignis-Bindings dynamisch durch den Ereignis-Producer bestimmt werden. Wir beginnen durch Schreiben einer abstrakten Unterklasse von `AnnotationLiteral`:

```
abstract class RoleBinding
   extends AnnotationLiteral<Role
>
   implements Role {}
```

Der Ereignis-Producer gibt eine Instanz dieser Klasse an `fire()` weiter:

```
documentEvent.fire( document, new RoleBinding() { public void value() { return user.getRole();
 } } );
```

9.5. Multiple Ereignis-Bindings

Typen von Ereignis-Bindings können kombiniert werden, zum Beispiel:

```
@Observable @Blog Event<Document
> blogEvent;
...
if (document.isBlog()) blogEvent.fire(document, new AnnotationLiteral<Updated
>(){});
```

Findet dieses Ereignis statt, so werden alle folgenden Observer-Methoden benachrichtigt:

```
public void afterBlogUpdate(@Observes @Updated @Blog Document document) { ... }
```

```
public void afterDocumentUpdate(@Observes @Updated Document document) { ... }
```

```
public void onAnyBlogEvent(@Observes @Blog Document document) { ... }
```

```
public void onAnyDocumentEvent(@Observes Document document) { ... }}}
```

9.6. Transaktionale Observer

Transaktionale Observers erhalten ihre Ereignisbenachrichtigungen vor oder nach der Abschlussphase der Transaktion während derer das Ereignis aufgegekommen ist. Zum Beispiel muss die folgende Observer-Methode einen Satz von Abfrageergebnissen neu laden, der im Applikationskontext gecacht ist, jedoch nur dann, wenn die den `Category`-Baum aktualisierenden Transaktionen erfolgreich sind:

```
public void refreshCategoryTree(@AfterTransactionSuccess @Observes CategoryUpdateEvent
 event) { ... }
```

Es gibt drei Arten von transaktionalen Observern:

- `@AfterTransactionSuccess`-Observers werden während der Abschlussphase der Transaktion aufgerufen, jedoch nur bei erfolgreichem Abschluss der Transaktion

- `@AfterTransactionFailure`-Observer werden während der Abschlussphase der Transaktion aufgerufen, jedoch nur, wenn der erfolgreiche Abschluss der Transaktion fehlschlägt
- `@AfterTransactionCompletion`-Observer werden während der Nach-Abschlussphase der Transaktion aufgerufen
- `@BeforeTransactionCompletion`-Observer werden während der Vor-Abschlussphase der Transaktion aufgerufen

Transaktional Observer sind in einem "stateful" Objektmodell wie Web Beans sehr wichtig, da der Status oft länger bestehen bleibt als eine einzelne atomare Transaktion.

Stellen wir uns vor, wir besitzen einen gecachten Satz von JPA-Abfrageergebnissen im Geltungsbereich der Anwendung:

```
@ApplicationScoped @Singleton
public class Catalog {

   @PersistenceContext EntityManager em;

   List<Product
> products;

   @Produces @Catalog
   List<Product
> getCatalog() {
      if (products==null) {
         products = em.createQuery("select p from Product p where p.deleted = false")
            .getResultList();
      }
      return products;
   }

}
```

Von Zeit zu Zeit wird ein `Product` erstellt oder gelöscht. Ist dies der Fall, so müssen wir den `Product`-Katalog neu laden. Wir sollten damit aber bis *nach* dem erfolgreichen Abschluss der Transaktion warten!

Das Web Bean, das `Products` erstellt oder löscht könnte Ereignisse aufrufen, zum Beispiel:

```
@Stateless
public class ProductManager {
```

```
    @PersistenceContext EntityManager em;
    @Observable Event<Product
> productEvent;

    public void delete(Product product) {
        em.delete(product);
        productEvent.fire(product, new AnnotationLiteral<Deleted
>(){});
    }

    public void persist(Product product) {
        em.persist(product);
        productEvent.fire(product, new AnnotationLiteral<Created
>(){});
    }

    ...

}
```

Und jetzt kann `Catalog` die Ereignisse nach erfolgreichem Abschluss der Transaktion beobachten:

```
@ApplicationScoped @Singleton
public class Catalog {

    ...

    void addProduct(@AfterTransactionSuccess @Observes @Created Product product) {
        products.add(product);
    }

    void addProduct(@AfterTransactionSuccess @Observes @Deleted Product product) {
        products.remove(product);
    }

}
```

Teil III. Das meiste aus starkem Tippen machen

Das zweite wichtige Thema von Web Beans ist *starke Typisierung* (sog. "strong Typing"). Die Informationen zu Abhängigkeiten, Interzeptoren und Dekoratoren eines Web Beans und die Informationen zu Ereigniskonsumenten (Event Consumers) für einen Ereignis-Producer sind in typensicheren Java-Konstrukten enthalten, die vom Kompilierer (Compiler) validiert werden können.

Sie sehen keine String-basierten Bezeichner in Web Beans Code; dies ist nicht der Fall weil das Framework diese unter Verwendung irgendwelcher Defaulting-Regeln # vor Ihnen verbirgt; sogenannte "Konfiguration nach Konvention" # sondern weil einfach keine Strings existieren!

Der offensichtliche Vorteil dieser Vorgehensweise ist, dass *jede* IDE Auto-Completion, Validierung und Refaktorierung ohne die Notwendigkeit spezieller Tools bereitstellen kann. Es existiert jedoch noch ein weiterer, nicht sofort ersichtlicher Vorteil. Es stellt sich nämlich heraus, dass Sie - wenn Sie über die Identifizierung von Objekten, Ereignissen oder Interzeptoren via Annotationen statt Namen nachdenken - Sie Gelegenheit haben, die semantische Ebene Ihres Code anzuheben.

Web Beans soll dazu ermutigen Annotationen zu entwickeln, die Konzepte formen, etwa

- `@Asynchronous`,
- `@Mock`,
- `@Secure` oder
- `@Updated`,

statt Namen wie Assoziationsbegriffe wie

- `asyncPaymentProcessor`,
- `mockPaymentProcessor`,
- `SecurityInterceptor` oder
- `DocumentUpdatedEvent` zu verwenden.

Die Annotationen sind wiederverwendbar. Sie helfen bei der Beschreibung gängiger Eigenschaften verschiedener Teile des Systems. Sie helfen uns bei der Kategorisierung und dem Verständnis unseres Codes. Sie helfen uns dabei auf gängige Weise mit gängigen Problemen umzugehen. Sie machen unseren Code leichter lesbar und einfacher zu verstehen.

Web Beans *Stereotypen* erweitern diese Idee um einen Schritt. Ein Stereotyp formt eine gängige *Rolle* in Ihrer Anwendungsarchitektur. Es enthält verschiedene Eigenschaften der Rolle,

einschließlich deren Geltungsbereich, Interzeptorbindungen, Deployment-Typ usw. in einem einzelnen, wiederverwendbaren Paket.

Sogar Web Beans XML Metadaten sind stark typisiert! Es gibt keinen Kompilierer für XML, daher nutzen Web Beans XML-Schemas zur Validierung der in XML vorkommenden Java-Typen und Attribute. Diese Vorgehensweise führt zur besseren Lesbarkeit der XML, ebenso wie Annotationen unseren Java Code einfacher lesbar machten.

Wir können jetzt einige fortgeschrittenere Features von Web Beans kennenlernen. Vergessen Sie nicht, dass diese Features unseren Code sowohl einfacher zu validieren als auch leserlicher machen sollen. Meist werden Sie diese Features nicht verwenden *müssen*, wenn Sie dies aber auf kluge Weise tun, so werden Sie deren Vorteile schnell zu schätzen wissen.

Stereotypen

Gemäß der Web Beans Spezifikation:

> In vielen Systemen produziert die Verwendung architektonischer Muster einen Satz wiederkehrender Web Bean Rollen. Ein Stereotyp gestattet dem Entwickler eines Frameworks die Identifizierung einer solchen Rolle und die Deklaration einiger gemeinsamer Metadaten für Web Beans mit dieser Rolle an einer zentralen Stelle.
>
> Ein Stereotyp beinhaltet eine beliebige Kombination von:
>
> - einem standardmäßigen Deployment-Typ,
> - einem standardmäßigen Geltungsbereich-Typ,
> - einer Einschränkung hinsichtlich des Geltungsbereichs des Web Beans,
> - einer Anforderung, dass das Web Bean einen bestimmten Typ implementiert oder erweitert und
> - einem Satz von Interzeptor Binding Annotationen.
>
> Ein Stereotyp kann auch festlegen, dass alle Web Beans mit dem Stereotyp standardmäßige Web Bean Namen besitzen.
>
> Ein Web Bean kann null, ein oder mehrere Stereotypen deklarieren.

Bei einem Stereotyp handelt es sich um einen Java Annotationstyp. Dieses Stereotyp identifiziert Action-Klassen in einem MVC-Framework:

```
@Retention(RUNTIME)
@Target(TYPE)
@Stereotype
public @interface Action {}
```

Wir verwenden das Stereotyp durch Anwendung der Annotation an einem Web Bean.

```
@Action
public class LoginAction { ... }
```

10.1. Standardmäßiger Geltungsbereich und Deployment-Typ für ein Stereotyp

Ein Stereotyp kann den standardmäßigen Geltungsbereich und/oder standardmäßigen Deployment-Typ für Web Beans mit diesem Stereotyp festlegen. Identifiziert der Deployment-Typ `@WebTier` etwa, dass Web Beans nur deployt werden sollten, wenn das System als eine Webanwendung ausgeführt wird, so könnten wird die folgenden Standards für Action-Klassen festlegen:

```
@Retention(RUNTIME)
@Target(TYPE)
@RequestScoped
@WebTier
@Stereotype
public @interface Action {}
```

Natürlich kann eine bestimmte Action diese Standards falls nötig immer noch außer Kraft setzen:

```
@Dependent @Mock @Action
public class MockLoginAction { ... }
```

Wenn wir alle Actions in einen bestimmten Geltungsbereich zwingen wollen, so können wir auch das tun.

10.2. Einschränkung des Geltungsbereichs und Typs mit einem Stereotyp

Nehmen wir an, wir wollten verhindern, dass Actions bestimmte Geltungsbereiche deklarieren. Web Beans lässt uns den Satz gestatteter Geltungsbereiche für Web Beans mit einem bestimmten Stereotyp explizit festlegen. Zum Beispiel:

```
@Retention(RUNTIME)
@Target(TYPE)
@RequestScoped
@WebTier
@Stereotype(supportedScopes=RequestScoped.class)
public @interface Action {}
```

Falls eine bestimmte Action-Klasse versucht einen anderen Geltungsbereich als den Anfragen-Geltungsbereich der Web Beans festzulegen, so wird zum Initialisierungszeitpunkt durch den Web Bean Manager eine Ausnahme gemeldet.

Wir können auch alle Web Beans mit einem bestimmten Stereotyp zur Implementierung eines Interface oder Erweiterung einer Klasse zwingen:

```
@Retention(RUNTIME)
@Target(TYPE)
@RequestScoped
@WebTier
@Stereotype(requiredTypes=AbstractAction.class)
public @interface Action {}
```

Falls eine bestimmte Action-Klasse die Klasse `AbstractAction` nicht erweitert, so wird zum Initialisierungszeitpunkt eine Ausnahme durch den Web Bean Manager gemeldet.

10.3. Interzeptor-Bindings für Stereotypen

Ein Stereotyp kann einen Satz von Interzeptor-Bindings festlegen, der an alle Web Beans mit diesem Stereotyp vererbt werden soll.

```
@Retention(RUNTIME)
@Target(TYPE)
@RequestScoped
@Transactional(requiresNew=true)
@Secure
@WebTier
@Stereotype
public @interface Action {}
```

Dies hilft uns einen weiteren Schritt weiter in Richtung der Trennung technischer Probleme und Business Code!

10.4. Namensstandardisierung und Stereotype

Zu guter Letzt können wir auch noch festlegen, dass alle Web Beans mit einem bestimmten Stereotyp einen Web Bean Namen besitzen, der vom Web Bean Manager standardisiert wird. Actions werden oft in JSP-Seiten referenziert, so dass sie den perfekten Anwendungsfall für dieses Feature darstellen. Alles, was wir tun müssen ist eine leere `@Named`-Annotation hinzuzufügen:

```
@Retention(RUNTIME)
@Target(TYPE)
@RequestScoped
@Transactional(requiresNew=true)
@Secure
@Named
@WebTier
@Stereotype
public @interface Action {}
```

Jetzt hat `LoginAction` den Namen `loginAction`.

10.5. Standard-Stereotypen

Wir haben bereits zwei Standard-Stereotypen kennengelernt, die durch die Web Beans Spezifikation definiert werden: `@Interceptor` und `@Decorator`.

Web Beans definiert einen weiteren Standard-Stereotyp:

```
@Named
@RequestScoped
@Stereotype
@Target({TYPE, METHOD})
@Retention(RUNTIME)
public @interface Model {}
```

Dieser Stereotyp ist für den Gebrauch mit JSF vorgesehen. Statt JSF gemanagte Beans zu verwenden, annotieren Sie ein Web Bean einfach `@Model`, und verwenden Sie es direkt in Ihrer JSF-Seite.

Specialization (Spezialisierung)

Wir haben bereits gesehen, wie das Web Beans Modell zur Dependency Einspeisung uns die *Außerkraftsetzung* der Implementierung eines API zum Zeitpunkt des Deployment gestattet. Das folgende Enterprise Web Bean zum Beispiel liefert eine Implementierung der API `PaymentProcessor` in Production:

```
@CreditCard @Stateless
public class CreditCardPaymentProcessor
      implements PaymentProcessor {
   ...
}
```

Aber in unserer Staging-Umgebung setzen wir diese Implementierung von `PaymentProcessor` mit einem anderen Web Bean außer Kraft:

```
@CreditCard @Stateless @Staging
public class StagingCreditCardPaymentProcessor
      implements PaymentProcessor {
   ...
}
```

Wir haben bei `StagingCreditCardPaymentProcessor` versucht, `AsyncPaymentProcessor` in einem bestimmten Deployment des Systems komplett zu ersetzen. In diesem Deployment, hätte der Deployment Typ `@Staging` eine höhere Priorität als der standardmäßige Deployment Typ `@Production` und daher Clients mit dem folgenden Einspeisungspunkt:

```
@CreditCard PaymentProcessor ccpp
```

Wir würden eine Instanz von `StagingCreditCardPaymentProcessor` erhalten.

Leider könnten wir in gleich mehrere Fallen tappen:

- Das Web Bean mit der höheren Priorität implementiert möglicherweise nicht alle API-Typen des Web Beans, das es außer Kraft zu setzen versucht,
- Das Web Bean mit der höheren Priorität deklariert möglicherweise nicht alle Binding-Typen des Web Beans, das es außer Kraft zu setzen versucht,
- Das Web Bean mit der höheren Priorität besitzt möglicherweise nicht denselben Namen wie das Web Bean, das es außer Kraft zu setzen versucht oder

- das Web Bean, das es außer Kraft zu setzen versucht deklariert möglicherweise eine Producer Methode, eine Bereinigungsmethode odereine Observer Methode.

In allen diesen Fällen kann das Web Bean, das wir außer Kraft zu setzen versucht haben, nach wie vor zur Runtime aufgerufen werden. Daher ist Außerkraftsetzung anfällig für Entwicklerfehler.

Web Beans bieten ein spezielles Feature namens *Specialization* (Spezialisierung), das dem Entwickler hilft, diese Stolperfallen zu umgehen. Specialization wirkt auf den ersten Blick etwas ungewöhnlich, ist aber in der Praxis einfach zu verwenden und Sie werden die zusätzliche Sicherheit, die es bietet bald schätzen.

11.1. Verwendung von Spezialisierung

Specialization ist ein Feature das spezifisch für einfache und Enterprise Web Beans ist. Um Specialization zu nutzen, muss ein Web Bean mit höherer Priorität:

- eine direkt Subklasse des Web Beans sein, das es außer Kraft setzt und
- ein einfaches Web Bean sein, falls das Web Bean, das es außer Kraft setzt ein einfaches Web Bean ist oder ein Enterprise Web Bean sein, falls das Web Bean, das es außer Kraft setzt ein Enterprise Web Bean ist und
- `@Specializes` annotiert sein.

```
@Stateless @Staging @Specializes
public class StagingCreditCardPaymentProcessor
      extends CreditCardPaymentProcessor {
   ...
}
```

Wir sprechen davon, dass das Web Bean mit höherer Priorität seine Superklasse *spezialisiert*.

11.2. Vorteile von Spezialisierung

Wann Spezialisierung eingesetzt wird:

- DIe Binding-Typen der Superklasse werden automatisch durch das mit `@Specializes` annotierte Web Bean geerbt und
- der Web Bean Name der Superklasse wird automatisch automatisch durch das mit `@Specializes` annotierte Web Bean geerbt und
- durch die Superklasse deklarierte Producer-Methoden, Bereinigungsmethoden und Observer-Methoden werden durch eine Instanz des mit `@Specializes` annotierten Web Beans aufgerufen.

In unserem Beispiel wird der Binding-Typ `@CreditCard` von `CreditCardPaymentProcessor` von `StagingCreditCardPaymentProcessor` geerbt.

Desweiteren validiert der Web Bean Manager dies:

- alle API-Typen der Superklasse sind API-Typen des mit `@Specializes` annotierten Web Beans (alle lokalen Interfaces der Superklasse Enterprise Bean sind auch lokale Interfaces der Subklasse),
- der Deployment-Typ des mit `@Specializes` annotierten Web Bean besitzt Vorrang vor dem Deployment-Typ der Superklasse und
- es existiert keun weiteres aktiviertes Web Bean, das ebenfalls die Superklasse spezialisiert.

Wird eine dieser Bedingungen verletzt, so meldet der Web Bean Manager zum Zeitpunkt der Initialisierung eine Ausnahme.

Wir können daher sicher sein, dass die Superklasse *nie* bei einem Deployment des Systems bei dem das mit `@Specializes` annotierte Web Bean deployt und aktiviert wird, aufgerufen wird.

Definition von Web Beans unter Verwendung von XML

Bis jetzt haben wir viele Beispiele von unter Verwendung von Annotationen deklarierten Web Beans gesehen. Es gibt jedoch ein paar Fälle, in denen keine Annotationen zur Definition des Web Beans verwendet werden können:

- Wenn die Implementierungsklasse von einer bereits bestehenden Bibliothek stammt oder
- wenn mehrere Web Beans mit derselben Implementierungsklasse existieren sollten.

In jedem dieser Fälle bieten uns Web Beans zwei Optionen:

- das Schreiben einer Producer-Methode oder
- das Deklarieren des Web Beans mittels XML.

Viele Frameworks verwenden XML zur Bereitstellung von Metadaten, die sich auf Java-Klassen beziehen. Web Beans jedoch verwenden eine sehr unterschiedliche Herangehensweise bei der Festlegung von Namen von Java-Klassen, Feldern oder Methoden als andere Frameworks. Statt dem Schreiben von Klassen- und Mitglieder-Namen als String-Werte von XML-Elementen und Attributen, gestatten Web Beans die Verwendung des Klassen- und Mitglieder-Namens als Name des XML-Elements.

Der Vorteil bei dieser Vorgehensweise ist, dass Sie ein XML-Schema schreiben können, dass Rechtschreibfehler in Ihrem XML-Dokument verhindert. Es ist sogar möglich, dass ein Tool das XML-Schema automatisch aus dem kompilierten Java-Code generiert. Oder eine integrierte Entwicklungsumgebung könnte dieselbe Validierung durchführen, ohne dass der explizite Generierungsschritt notwendig wäre.

12.1. Deklaration von Web Bean Klassen

Für jedes Java-Paket definieren Web Beans einen entsprechenden XML-Namespace. Der Namespace wird durch Voranstellen von `urn:java:` vor den Java-Paketnamen gebildet. Für das Paket `com.mydomain.myapp` ist der XML-Namespace `urn:java:com.mydomain.myapp`.

Auf Java-Typen, die zu einem Paket gehören, wird verwiesen, indem ein XML-Element in dem dem Paket entsprechenden Namespace verwendet wird. Derr Name des Elements ist der Name des Java-Typs. Felder und Methoden des Typs werden durch untergeordnete Elemente in demselben Namespace festgelegt. Handelt es sich bei dem Typ um eine Annotation, so werden Mitglieder durch Attribute des Elements festgelegt.

Zum Beispiel bezieht sich das Element `<util:Date/>` im folgenden XML-Fragment auf die Klasse `java.util.Date`:

```
<WebBeans xmlns="urn:java:javax.webbeans"
```

```
        xmlns:util="urn:java:java.util">

    <util:Date/>

</WebBeans
>
```

Und das ist alles an Code was wir benötigen, um zu deklarieren dass es sich bei `Date` um ein einfaches Web Bean handelt! Eine Instanz von `Date` kann jetzt in ein beliebiges anderes Web Bean eingespeist werden:

```
@Current Date date
```

12.2. Deklaration von Web Bean Metadaten

Wir können Geltungsbereich, Deployment-Typ und Interzeptor Binding-Typen deklarieren, indem wir direkte untergeordnete Elemente der Web Bean Dellaration verwenden:

```
<myapp:ShoppingCart>
    <SessionScoped/>
    <myfwk:Transactional requiresNew="true"/>
    <myfwk:Secure/>
</myapp:ShoppingCart
>
```

Wir verwenden exakt dieselbe Vorgehensweise, um Namen und Binding-Typ festzulegen:

```
<util:Date>
    <Named
>currentTime</Named>
</util:Date>

<util:Date>
    <SessionScoped/>
    <myapp:Login/>
    <Named
>loginTime</Named>
</util:Date>

<util:Date>
```

```
   <ApplicationScoped/>
   <myapp:SystemStart/>
   <Named
>systemStartTime</Named>
</util:Date
>
```

Wobei `@Login` und `@SystemStart` Binding Annotationstypen sind.

```
@Current Date currentTime;
@Login Date loginTime;
@SystemStart Date systemStartTime;
```

Wie gewöhnlich kann ein Web Bean mehrere Binding-Typen unterstützen:

```
<myapp:AsynchronousChequePaymentProcessor>
   <myapp:PayByCheque/>
   <myapp:Asynchronous/>
</myapp:AsynchronousChequePaymentProcessor
>
```

Interzeptoren und Dekoratoren sind nur einfache Web Beans und können daher wie jedes andere einfache Web Bean deklariert werden:

```
<myfwk:TransactionInterceptor>
   <Interceptor/>
   <myfwk:Transactional/>
</myfwk:TransactionInterceptor
>
```

12.3. Deklaration von Web Bean Mitgliedern

TODO!

12.4. Deklaration von inline Web Beans

Web Beans lassen uns ein Web Bean an einem Einspeisungspunkt definieren. Zum Beispiel:

```
<myapp:System>
```

```
   <ApplicationScoped/>
   <myapp:admin>
      <myapp:Name>
         <myapp:firstname
>Gavin</myapp:firstname>
         <myapp:lastname
>King</myapp:lastname>
         <myapp:email
>gavin@hibernate.org</myapp:email>
      </myapp:Name>
   </myapp:admin>
</myapp:System
>
```

Das `<Name>`-Element deklariert ein einfaches Web Bean von Geltungsbereich `@Dependent` und Klasse `Name` mit einem Satz anfänglicher Feldwerte. Dieses Web Bean besitzt ein spezielles, Container-generiertes Binding und ist daher nur an dem spezifischen Einspeisungspunkt einspeisbar, an dem es deklariert wird.

Dieses einfache aber leistungsfähige Feature gestattet die Verwendung des Web Beans XML-Formats zur Festlegung ganzer Diagramme von Java-Objekten. Es ist noch keine volndige datenbindende Lösung, aber ganz nah dran!

12.5. Verwendung eines Schemas

Wenn Personen, die keine Java-Entwickler sind, Autoren unseres XML-Dokumentformats sein sollen oder diese keinen Zugriff auf unseren Code haben, so müssen wir ein Schema bereitstellen. Es gibt nichts, was hinsichtlich des Schreibens oder der Verwendung des Schemas spezifisch für Web Beans ist.

```
<WebBeans xmlns="urn:java:javax.webbeans"
      xmlns:myapp="urn:java:com.mydomain.myapp"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="urn:java:javax.webbeans http://java.sun.com/jee/web-beans-1.0.xsd
                        urn:java:com.mydomain.myapp http://mydomain.com/xsd/myapp-1.2.xsd">

   <myapp:System>
      ...
   </myapp:System>

</WebBeans
>
```

Das Schreiben eines XML-Schemas ist recht mühselig. Daher liefert das Web Beans RI Projekt ein Tool, das automatisch das XML-Schema aus kompiliertem Java-Code generiert.

Teil IV. Web Beans und das Java EE-Ã#kosystem

Das dritte Thema von Web Beans ist die *Integration*. Web Beans wurden derart entwickelt, dass sie mit anderer Technologie zusammenarbeiten und dem Anwendungsentwickler dabei helfen, diese andere Technologie zusammenzufügen. Bei Web Beans handelt es sich um offene Technologie. Sie bilden einen Teil des Java EE Ökosystems und sind selbst die Grundlage für ein neues Ökosystem portabler Erweiterungen und Integration mit bestehenden Frameworks und bestehender Technologie.

Wir haben bereits gesehen, wie Web Beans bei der Integration von EJB und JSF helfen, indem sie gestatten, dass EJBs direkt an JSF-Seiten gebunden werden. Das ist nur der Anfang. Web Beans bieten dasselbe Potential zur Diversifizierung anderer Technologien, wie etwa Business Process Management Engines, anderer Web Frameworks und Komponentenmodellen Dritter. Die Java EE Plattform wird nie zur Standardisierung aller interessanter Technologien in der Lage sein, die bei der Entwicklung von Java-Anwendungen verwendet werden, aber Web Beans vereinfachen die nahtlose Verwendung solcher Technologien, die noch nicht Teil der Plattform sind, innerhalb einer Java EE Umgebung.

Wir wollen Ihnen jetzt zeigen, wie Sie die Java EE platform in einer Web Beans verwendenden Anwendung am besten nutzen. Wir sehen uns auch kurz einen Satz von SPIs an, die portable Erweiterungen zu Web Beans unterstützen sollen. Sie werden diese SPIs vielleicht nie direkt benutzen müssen, aber es ist hilfreich diese zu kennen, falls Sie sie einmal brauchen. Primär ist wichtig, dass Sie diese bei jeder Verwendung von Erweiterungen Dritter indirekt nutzen können.

Java EE Integration

Web Beans sind voll in die Java EE Umgebung integriert. Web Beans besitzen Zugriff auf Java EE Resourcen und JPA Persistenzkontexte. Sie können in Unified EL Ausdrücken in JSF- und JSP-Seiten verwendet werden. Sie können sogar in einige Objekte eingespeist werden, etwa Servlets and Message-Driven Beans, die keine Web Beans sind.

13.1. Einspeisung von Java EE Ressourcen in ein Web Bean

Alle einfachen wie auch Enterprise Web Beans können die Java EE "Dependency"-Einspeisung mittels `@Resource`, `@EJB` und `@PersistenceContext` verwenden. Wir haben bereits einige Beispiele hierfür gesehen, obwohl wir diesen zum damaligen Zeitpunkt nicht viel Beachtung geschenkt haben:

```
@Transactional @Interceptor
public class TransactionInterceptor {

   @Resource Transaction transaction;

   @AroundInvoke public Object manageTransaction(InvocationContext ctx) { ... }

}
```

```
@SessionScoped
public class Login {

   @Current Credentials credentials;
   @PersistenceContext EntityManager userDatabase;

   ...

}
```

Die Java EE `@PostConstruct` und `@PreDestroy` Callbacks werden ebenfalls für alle einfachen wie auch Enterprise Web Beans unterstützt. Die `@PostConstruct`-Methode wird nach Durchführung *aller* Einspeisungen aufgerufen.

Es gilt eine Einschränkung hier: `@PersistenceContext(type=EXTENDED)` wird nicht für einfache Web Beans unterstützt.

13.2. Aufruf eines Web Bean von einem Servlet

In Java EE 6 ist die Verwendung eines Web Beans von einem Servlet ganz einfach. Speisen Sie einfach das Web Bean mittels Web Beans Field oder Initialisierungsmethodeneinspeisung (sog. "Initializer Method Injection") ein.

```
public class Login extends HttpServlet {

   @Current Credentials credentials;
   @Current Login login;

   @Override
   public void service(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
      credentials.setUsername( request.getAttribute("username") ):
      credentials.setPassword( request.getAttribute("password") ):
      login.login();
      if ( login.isLoggedIn() ) {
         response.sendRedirect("/home.jsp");
      }
      else {
         response.sendRedirect("/loginError.jsp");
      }
   }

}
```

Der Web Beans Client Proxy kümmert sich um Aufrufe der Routing-Methode vom Servlet, um die Instanzen von `Credentials` und `Login` für die aktuelle Anfrage und HTTP-Session zu korrigieren.

13.3. Aufruf eines Web Beans von einem Message-Driven Bean

Einspeisung von Web Beans gilt für alle EJBs, selbst wenn sie nicht der Steuerung des Web Bean Manager s unterliegen (wenn sie etwa durch direkten JNDI-Lookup oder Einspeisung mittels `@EJB` erworben wurden). Insbesondere Web Beans Einspeisung in Message-Driven Beans, die nicht als Web Beans angesehen werden, da sie nicht eingespeist werden können.

Sie können sogar Web Beans Interzeptor-Bindings für Message-Driven Beans.

```
@Transactional @MessageDriven
public class ProcessOrder implements MessageListener {
```

```
   @Current Inventory inventory;
   @PersistenceContext EntityManager em;

   public void onMessage(Message message) {
      ...
   }

}
```

Daher ist der Empfang von Nachrichten in einer Web Beans Umgebung sehr einfach. Seien Sie sich aber dessen bewusst, dass bei Lieferung einer Nachricht an ein Message-Driven Bean keine Session oder Konversationskontext verfügbar ist. Es sind nur `@RequestScoped` und `@ApplicationScoped` Web Beans verfügbar.

Es ist ebenfalls ganz einfach mittels Web Beans Nachrichten zu versenden.

13.4. JMS Endpunkte

Das Versenden von Nachrichten unter Verwendung von JMS kann aufgrund der Anzahl verschiedener Objekte mit denen Sie zu tun haben recht komplex sein. Für Warteschlangen haben wir `Queue`, `QueueConnectionFactory`, `QueueConnection`, `QueueSession` und `QueueSender`. Für Topics haben wir `Topic`, `TopicConnectionFactory`, `TopicConnection`, `TopicSession` und `TopicPublisher`. Jedes dieser Objekte besitzt einen eigenen Lebenszyklus und ein eigenes Threading-Modell, das unsere Aufmerksamkeit erfordert.

Web Beans übernehmen all das für uns. Das Einzige, was wir tun müssen ist unsere Warteschlange oder unser Topic in `web-beans.xml` zu deklarieren und einen assoziierten Binding-Typ und eine Connection-Factory festzulegen.

```
<Queue>
   <destination
>java:comp/env/jms/OrderQueue</destination>
   <connectionFactory
>java:comp/env/jms/QueueConnectionFactory</connectionFactory>
   <myapp:OrderProcessor/>
</Queue
>
```

```
<Topic>
   <destination
>java:comp/env/jms/StockPrices</destination>
   <connectionFactory
```

```
>java:comp/env/jms/TopicConnectionFactory</connectionFactory>
   <myapp:StockPrices/>
</Topic
>
```

Jetzt können wir einfach `Queue`, `QueueConnection`, `QueueSession` oder `QueueSender` für eine Warteschlange oder aber `Topic`, `TopicConnection`, `TopicSession` oder `TopicPublisher` für ein Topic einspeisen.

```
@OrderProcessor QueueSender orderSender;
@OrderProcessor QueueSession orderSession;

public void sendMessage() {
   MapMessage msg = orderSession.createMapMessage();
   ...
   orderSender.send(msg);
}
```

```
@StockPrices TopicPublisher pricePublisher;
@StockPrices TopicSession priceSession;

public void sendMessage(String price) {
   pricePublisher.send( priceSession.createTextMessage(price) );
}
```

Der Lebenszyklus der eingespeisten JMS-Objekte wird komplett durch den Web Bean Manager gesteuert.

13.5. Packen und Deployment

Web Beans definiert kein spezielles Deployment-Archiv. Sie können Web Beans in JARs, EJB-JARs oder WARs # verpacken, jedem Deployment Speicherort im Klassenpfad der Anwendung. Allerdings muss jedes Web Beans enthaltene Archiv eine Datei namens `web-beans.xml` im `META-INF` oder `WEB-INF`-Verzeichnis enthalten. Die Datei kann leer sein. Web Beans die in Archiven deployt werden, die keine `web-beans.xml`-Datei enthalten, sind nicht für die Verwendung in der Anwendung verfügbar.

Für die Java SE Ausführung können Web Beans an einem beliebigen Speicherort deployt werden, in welchem EJBs zur Ausführung durch den einbettbaren EJB Lite Container deployt werden können. Auch hier muss jeder Speicherort eine `web-beans.xml`-Datei enthalten.

Erweiterung von Web Beans

Web Beans sind als Plattform für Frameworks, Erweiterungen und Integration mit anderer Technologie vorgesehen. Web Beans bieten daher einen Satz von SPIs für den Gebrauch durch Entwickler übertragbarer Erweiterungen zu Web Beans. Die folgenden Arten von Erweiterungen zum Beispiel wurden von den Designern von Web Beans vorgesehen:

- Integration mit Business Process Management Engines,
- Integration mit den Frameworks Dritter, wie etwa Spring, Seam, GWT oder Wicket und
- neue, auf dem Web Beans Programmiermodell basierende Technologie.

Der zentrale Kern zur Erweiterung von Web Beans ist das `Manager`-Objekt.

14.1. Das `Manager`-Objekt

Das `Manager`-Interface die programmatische Registrierung und den Erhalt von Web Beans, Interzeptoren, Dekoratoren, Observern und Kontexten.

```
public interface Manager
{

  public <T
> Set<Bean<T
>
> resolveByType(Class<T
> type, Annotation... bindings);

  public <T
> Set<Bean<T
>
> resolveByType(TypeLiteral<T
> apiType,
      Annotation... bindings);

  public <T
> T getInstanceByType(Class<T
> type, Annotation... bindings);

  public <T
> T getInstanceByType(TypeLiteral<T
> type,
      Annotation... bindings);
```

```
   public Set<Bean<?>
> resolveByName(String name);

   public Object getInstanceByName(String name);

   public <T
> T getInstance(Bean<T
> bean);

   public void fireEvent(Object event, Annotation... bindings);

   public Context getContext(Class<? extends Annotation
> scopeType);

   public Manager addContext(Context context);

   public Manager addBean(Bean<?> bean);

   public Manager addInterceptor(Interceptor interceptor);

   public Manager addDecorator(Decorator decorator);

   public <T
> Manager addObserver(Observer<T
> observer, Class<T
> eventType,
         Annotation... bindings);

   public <T
> Manager addObserver(Observer<T
> observer, TypeLiteral<T
> eventType,
         Annotation... bindings);

   public <T
> Manager removeObserver(Observer<T
> observer, Class<T
> eventType,
         Annotation... bindings);

   public <T
> Manager removeObserver(Observer<T
> observer,
         TypeLiteral<T
```

```
> eventType, Annotation... bindings);

   public <T
> Set<Observer<T
>
> resolveObservers(T event, Annotation... bindings);

   public List<Interceptor
> resolveInterceptors(InterceptionType type,
         Annotation... interceptorBindings);

   public List<Decorator
> resolveDecorators(Set<Class<?>
> types,
         Annotation... bindings);

}
```

Wir können eine Instanz von `Manager` via Einspeisung erhalten:

```
@Current Manager Manager
```

14.2. Die `Bean`-Klasse

Instanzen der abstrakten Klasse `Bean` repräsentieren Web Beans. Für jedes Web Bean in der Anwendung wird eine Instanz von `Bean` mit dem `Manager`-Objekt registriert.

```
public abstract class Bean<T> {

   private final Manager manager;

   protected Bean(Manager manager) {
      this.manager=manager;
   }

   protected Manager getManager() {
      return manager;
   }

   public abstract Set<Class> getTypes();
   public abstract Set<Annotation> getBindingTypes();
   public abstract Class<? extends Annotation> getScopeType();
```

```
   public abstract Class<? extends Annotation> getDeploymentType();
   public abstract String getName();

   public abstract boolean isSerializable();
   public abstract boolean isNullable();

   public abstract T create();
   public abstract void destroy(T instance);

}
```

Es ist möglich, die `Bean`-Klasse zu erweitern und Instanzen durch Aufruf von `Manager.addBean()` zu registrieren, um Support für neue Arten von Web Beans zu bieten, neben denen, die durch die Web Beans Spezifikation definiert sind (einfache und Enterprise Web Beans, Producer Methoden und JMS Endpunkte). Zum Beispiel könnten wir die `Bean`-Klasse verwenden, um zu ermöglichen, dass durch ein anderes Framework gemanagte Objekte in Web Beans eingespeist werden.

Durch die Web Beans Spezifikation werden zwei Unterklassen von `Bean` definiert: `Interceptor` und `Decorator`.

14.3. Das `Context`-Interface

Das `Context`-Interface unterstützt die Hinzufügung neuer Geltungsbereiche zu Web Beans oder die Erweiterung eingebauter Geltungsbereiche zu neuen Umgebungen.

```
public interface Context {

   public Class<? extends Annotation> getScopeType();

   public <T> T get(Bean<T> bean, boolean create);

   boolean isActive();

}
```

Wir könnten zum Beispiel `Context` implementieren, um den Geltungsbereich eines Business Prozesses zu Web Beans oder Support für den Konversationsgeltungsbereich einer Wickets verwendenen Anwendung hinzuzufügen.

Die nächsten Schritte

Da Web Beans so neu sind, existieren noch nicht so viele Informationen online.

Natürlich ist die Web Beans Spezifikation die beste Quelle für weitere Information zu Web Beans. Die Spezifikation umfasst rund 100 Seiten, ist also nur etwa doppelt so lang wie dieser Artikel, aber nahezu so einfach verständlich wie dieser. Sie enthält aber zahlreiche Einzelheiten, auf die hier nicht eingegangen wird. Die Spezifikation ist unter `http://jcp.org/en/jsr/detail?id=299` verfügbar.

Die Web Beans Referenzimplementierung wird unter `http://seamframework.org/WebBeans` entwickelt. Das RI Entwicklungsteam und die Web Beans Spezifikation führen unter `http://in.relation.to` einen Blog. Dieser Artikel basiert im Wesentlichen auf eine dort veröffentlichte Reihe von Blog-Einträgen.

Teil V. Web Beans Reference

Web Beans is the reference implementation of JSR-299, and is used by JBoss AS and Glassfish to provide JSR-299 services for Java Enterprise Edition applications. Web Beans also goes beyond the environments and APIs defined by the JSR-299 specification and provides support for a number of other environments (such as a servlet container such as Tomcat, or Java SE) and additional APIs and modules (such as logging, XSD generation for the JSR-299 XML deployment descriptors).

If you want to get started quickly using Web Beans with JBoss AS or Tomcat and experiment with one of the examples, take a look at *Kapitel 3, Getting started with Web Beans, the Reference Implementation of JSR-299*. Otherwise read on for a exhaustive discussion of using Web Beans in all the environments and application servers it supports, as well the Web Beans extensions.

Application Servers and environments supported by Web Beans

16.1. Using Web Beans with JBoss AS

No special configuration of your application, beyond adding either `META-INF/beans.xml` or `WEB-INF/beans.xml` is needed.

If you are using JBoss AS 5.0.1.GA then you'll need to install Web Beans as an extra. First we need to tell Web Beans where JBoss is located. Edit `jboss-as/build.properties` and set the `jboss.home` property. For example:

```
jboss.home=/Applications/jboss-5.0.1.GA
```

Now we can install Web Beans:

```
$ cd webbeans-$VERSION/jboss-as
$ ant update
```

Anmerkung

A new deployer, `webbeans.deployer` is added to JBoss AS. This adds supports for JSR-299 deployments to JBoss AS, and allows Web Beans to query the EJB3 container and discover which EJBs are installed in your application.

Web Beans is built into all releases of JBoss AS from 5.1 onwards.

16.2. Glassfish

TODO

16.3. Servlet Containers (such as Tomcat or Jetty)

Web Beans can be used in any Servlet container such as Tomcat 6.0 or Jetty 6.1.

Anmerkung

Web Beans doesn't support deploying session beans, injection using `@EJB`, or `@PersistenceContext` or using transactional events in Servlet containers.

Web Beans should be used as a web application library in a servlet container. You should place `webbeans-servlet.jar` in `WEB-INF/lib`. `webbeans-serv;et.jar` is an "uber-jar" provided for your convenience. Instead, you could use its component jars:

- `jsr299-api.jar`
- `webbeans-api.jar`
- `webbeans-spi.jar`
- `webbeans-core.jar`
- `webbeans-logging.jar`
- `webbeans-servlet-int.jar`
- `javassist.jar`
- `dom4j.jar`

You also need to explicitly specify the servlet listener (used to boot Web Beans, and control its interaction with requests) in `web.xml`:

```
<listener>
  <listener-class>org.jboss.webbeans.environment.servlet.Listener</listener-class>
</listener>
```

16.3.1. Tomcat

Tomcat has a read-only JNDI, so Web Beans can't automatically bind the Manager. To bind the Manager into JNDI, you should add the following to your `META-INF/context.xml`:

```
<Resource name="app/Manager"
        auth="Container"
        type="javax.inject.manager.Manager"
        factory="org.jboss.webbeans.resources.ManagerObjectFactory"/>
```

and make it available to your deployment by adding this to `web.xml`:

```
<resource-env-ref>
  <resource-env-ref-name>
    app/Manager
  </resource-env-ref-name>
  <resource-env-ref-type>
    javax.inject.manager.Manager
  </resource-env-ref-type>
</resource-env-ref>
```

Tomcat only allows you to bind entries to `java:comp/env`, so the Manager will be available at `java:comp/env/app/Manager`

Web Beans also supports Servlet injection in Tomcat. To enable this, place the `webbeans-tomcat-support.jar` in `$TOMCAT_HOME/lib`, and add the following to your `META-INF/context.xml`:

```
<Listener className="org.jboss.webbeans.environment.tomcat.WebBeansLifecycleListener" />
```

16.4. Java SE

Apart from improved integration of the Enterprise Java stack, Web Beans also provides a state of the art typesafe, stateful dependency injection framework. This is useful in a wide range of application types, enterprise or otherwise. To facilitate this, Web Beans provides a simple means for executing in the Java Standard Edition environment independently of any Enterprise Edition features.

When executing in the SE environment the following features of Web Beans are available:

- `Simple Web Beans (POJOs)`
- `Typesafe Dependency Injection`
- `Application and Dependent Contexts`
- `Binding Types`
- `Stereotypes`
- `Typesafe Event Model`

16.4.1. Web Beans SE Module

To make life easy for developers Web Beans provides a special module with a main method which will boot the Web Beans manager, automatically registering all simple Web Beans found on the classpath. This eliminates the need for application developers to write any bootstrapping

code. The entry point for a Web Beans SE applications is a simple Web Bean which observes the standard `@Deployed Manager` event. The command line paramters can be injected using either of the following:

```
@Parameters List<String> params;
@Parameters String[] paramsArray; // useful for compatability with existing classes
```

Here's an example of a simple Web Beans SE application:

```
@ApplicationScoped
public class HelloWorld
{
   @Parameters List<String> parameters;

   public void printHello( @Observes @Deployed Manager manager )
   {
      System.out.println( "Hello " + parameters.get(0) );
   }
}
```

Web Beans SE applications are started by running the following main method.

```
java org.jboss.webbeans.environments.se.StartMain <args>
```

If you need to do any custom initialization of the Web Beans manager, for example registering custom contexts or initializing resources for your beans you can do so in response to the `@Initialized Manager` event. The following example registers a custom context:

```
public class PerformSetup
{

   public void setup( @Observes @Initialized Manager manager )
   {
      manager.addContext( ThreadContext.INSTANCE );
   }
}
```

Anmerkung

The command line parameters do not become available for injection until the `@Deployed Manager` event is fired. If you need access to the parameters during initialization you can do so via the `public static String getParameters()` method in `StartMain`.

JSR-299 extensions available as part of Web Beans

Wichtig

These modules are usable on any JSR-299 implementation, not just Web Beans!

17.1. Web Beans Logger

Adding logging to your application is now even easier with simple injection of a logger object into any JSR-299 bean. Simply annotate a org.jboss.webbeans.log.Log type member with *@Logger* and an appropriate logger object will be injected into any instance of the bean.

```
public class Checkout {
   import org.jboss.webbeans.annotation.Logger;
   import org.jboss.webbeans.log.Log;

   @Logger
   private Log log;

   void invoiceItems() {
      ShoppingCart cart;
      . . .
      log.debug("Items invoiced for {0}", cart);
   }

}
```

The example shows how objects can be interpolated into a message. This interpolation is done using *java.text.MessageFormat*, so see the JavaDoc for that class for more details. In this case, the ShoppingCart should have implemented the *toString()* method to produce a human readable value that is meaningful in messages. Normally, this call would have involved evaluating cart.toString() with String concatenation to produce a single String argument. Thus it was necessary to surround the call with an if-statement using the condition *log.isDebugEnabled()* to avoid the expensive String concatenation if the message was not actually going to be used. However, when using @Logger injected logging, the conditional test can be left out since the object arguments are not evaluated unless the message is going to be logged.

Anmerkung

You can add the Web Beans Logger to your project by including webbeans-logger.jar and webbeans-logging.jar to your project. Alternatively, express a dependency on the `org.jboss.webbeans:webbeans-logger` Maven artifact.

If you are using Web Beans as your JSR-299 implementation, there is no need to include `webbeans-logging.jar` as it's already included.

Alternative view layers

18.1. Using Web Beans with Wicket

18.1.1. The `WebApplication` class

Each wicket application must have a `WebApplication` subclass; Web Beans provides, for your utility, a subclass of this which sets up the Wicket/JSR-299 integration. You should subclass `org.jboss.webbeans.wicket.WebBeansApplication`.

Anmerkung

If you would prefer not to subclass `WebBeansApplication`, you can manually add a (small!) number of overrides and listeners to your own `WebApplication` subclass. The javadocs of `WebBeansApplication`detail this.

For example:

```
public class SampleApplication extends WebBeansApplication {
   @Override
   public Class getHomePage() {
      return HomePage.class;
   }
}
```

18.1.2. Conversations with Wicket

The conversation scope can be used in Web Beans with the Apache Wicket web framework, through the `webbeans-wicket` module. This module takes care of:

- Setting up the conversation context at the beginning of a Wicket request, and tearing it down afterwards
- Storing the id of any long-running conversation in Wicket's metadata when the page response is complete
- Activating the correct long-running conversation based upon which page is being accessed
- Propagating the conversation context for any long-running conversation to new pages

18.1.2.1. Starting and stopping conversations in Wicket

As JSF applications, a conversation *always* exists for any request, but its lifetime is only that of the current request unless it is marked as *long-running*. For Wicket applications this is

accomplished as in JSF applications, by injecting the `@Current Conversation` and then invoking `conversation.begin()`. Likewise, conversations are ended with `conversation.end()`

18.1.2.2. Long running conversation propagation in Wicket

When a conversation is marked as long-running, the id of that conversation will be stored in Wicket's metadata for the current page. If a new page is created and set as the response target through `setResponsePage`, this new page will also participate in this conversation. This occurs for both directly instantiated pages (`setResponsePage(new OtherPage())`), as well as for bookmarkable pages created with `setResponsePage(OtherPage.class)` where `OtherPage.class` is mounted as bookmarkable from your `WebApplication` subclass (or through annotations). In the latter case, because the new page instance is not created until after a redirect, the conversation id will be propagated through a request parameter, and then stored in page metadata after the redirect.

Anhang A. Integrating Web Beans into other environments

Currently Web Beans only runs in JBoss AS 5; integrating the RI into other EE environments (for example another application server like Glassfish), into a servlet container (like Tomcat), or with an Embedded EJB3.1 implementation is fairly easy. In this Appendix we will briefly discuss the steps needed.

A.1. The Web Beans SPI

The Web Beans SPI is located in the `webbeans-spi` module, and packaged as `webbeans-spi.jar`. Some SPIs are optional, if you need to override the default behavior, others are required.

All interfaces in the SPI support the decorator pattern and provide a `Forwarding` class located in the `helpers` sub package. Additional, commonly used, utility classes, and standard implementations are also located in the `helpers` sub package.

A.1.1. Web Bean Discovery

```
/**
   * Gets list of all classes in classpath archives with META-INF/beans.xml (or
   * for WARs WEB-INF/beans.xml) files
   *
   * @return An iterable over the classes
   */
  public Iterable<Class<?>> discoverWebBeanClasses();

  /**
   * Gets a list of all deployment descriptors in the app classpath
   *
   * @return An iterable over the beans.xml files
   */
  public Iterable<URL> discoverWebBeansXml();
```

The discovery of Web Bean classes and `beans.xml` files is self-explanatory (the algorithm is described in Section 11.1 of the JSR-299 specification, and isn't repeated here).

A.1.2. EJB services

Anmerkung

Web Beans will run without an EJB container; in this case you don't need to implement the EJB SPI.

Web Beans also delegates EJB3 bean discovery to the container so that it doesn't have to scan for EJB3 annotations or parse `ejb-jar.xml`. For each EJB in the application an EJBDescriptor should be discovered:

```
public interface EjbDescriptor<T>
{

  /**
   * Gets the EJB type
   *
   * @return The EJB Bean class
   */
  public Class<T> getType();

  /**
   * Gets the local business interfaces of the EJB
   *
   * @return An iterator over the local business interfaces
   */
  public Iterable<BusinessInterfaceDescriptor<?>> getLocalBusinessInterfaces();

  /**
   * Gets the remote business interfaces of the EJB
   *
   * @return An iterator over the remote business interfaces
   */
  public Iterable<BusinessInterfaceDescriptor<?>> getRemoteBusinessInterfaces();

  /**
   * Get the remove methods of the EJB
   *
   * @return An iterator over the remove methods
   */
  public Iterable<Method> getRemoveMethods();

  /**
```

```
    * Indicates if the bean is stateless
    *
    * @return True if stateless, false otherwise
    */
   public boolean isStateless();

   /**
    * Indicates if the bean is a EJB 3.1 Singleton
    *
    * @return True if the bean is a singleton, false otherwise
    */
   public boolean isSingleton();

   /**
    * Indicates if the EJB is stateful
    *
    * @return True if the bean is stateful, false otherwise
    */
   public boolean isStateful();

   /**
    * Indicates if the EJB is and MDB
    *
    * @return True if the bean is an MDB, false otherwise
    */
   public boolean isMessageDriven();

   /**
    * Gets the EJB name
    *
    * @return The name
    */
   public String getEjbName();
```

Der `EjbDescriptor` ist recht leicht verständlich und sollte relevante Metadaten wie in der EJB-Spezifikation definiert wiedergeben. Neben diesen beiden Interfaces existiert ein `BusinessInterfaceDescriptor`, der ein lokales Business-Interface repräsentiert (die Interface-Klasse und den für die Suche einer Instanz des EJB verwendeten jndi-Namens enthaltend).

The resolution of `@EJB` (for injection into simple beans), the resolution of local EJBs (for backing session beans) and remote EJBs (for injection as a Java EE resource) is delegated to the container. You must provide an implementation of `org.jboss.webbeans.ejb.spi.EjbServices` which provides these operations. For resolving the `@EJB` injection point, Web Beans will provide the

`InjectionPoint`; for resolving local EJBs, the `EjbDescriptor` will be provided, and for remote EJBs the `jndiName`, `mappedName`, or `ejbLink` will be provided.

When resolving local EJBs (used to back session beans) a wrapper (`SessionObjectReference`) around the EJB reference is returned. This wrapper allows Web Beans to request a reference that implements the given business interface, and, in the case of SFSBs, request the removal of the EJB from the container.

A.1.3. JPA services

Just as EJB resolution is delegated to the container, resolution of `@PersistenceContext` for injection into simple beans (with the `InjectionPoint` provided), and resolution of persistence contexts and persistence units (with the `unitName` provided) for injection as a Java EE resource is delegated to the container.

To allow JPA integration, the `JpaServices` interface should be implemented.

Web Beans also needs to know what entities are in a deployment (so that they aren't managed by Web Beans). An implementation that detects entities through `@Entity` and `orm.xml` is provided by default. If you want to provide support for a entities defined by a JPA provider (such as Hibernate's `.hbm.xml` you can wrap or replace the default implementation.

```
EntityDiscovery delegate = bootstrap.getServices().get(EntityDiscovery.class);
```

A.1.4. Transaction Services

Web Beans must delegate JTA activities to the container. The SPI provides a couple hooks to easily achieve this with the `TransactionServices` interface.

```
public interface TransactionServices
{
  /**
   * Possible status conditions for a transaction. This can be used by SPI
   * providers to keep track for which status an observer is used.
   */
  public static enum Status
  {
    ALL, SUCCESS, FAILURE
  }

  /**
   * Registers a synchronization object with the currently executing
   * transaction.
   *
```

```
    * @see javax.transaction.Synchronization
    * @param synchronizedObserver
    */
   public void registerSynchronization(Synchronization synchronizedObserver);

   /**
    * Queries the status of the current execution to see if a transaction is
    * currently active.
    *
    * @return true if a transaction is active
    */
   public boolean isTransactionActive();
}
```

The enumeration `Status` is a convenience for implementors to be able to keep track of whether a synchronization is supposed to notify an observer only when the transaction is successful, or after a failure, or regardless of the status of the transaction.

Any `javax.transaction.Synchronization` implementation may be passed to the `registerSynchronization()` method and the SPI implementation should immediately register the synchronization with the JTA transaction manager used for the EJBs.

To make it easier to determine whether or not a transaction is currently active for the requesting thread, the `isTransactionActive()` method can be used. The SPI implementation should query the same JTA transaction manager used for the EJBs.

A.1.5. JMS services

A number of JMS operations are not container specific, and so should be provided via the SPI `JmsServices`. JMS does not specify how to obtain a `ConnectionFactory` so the SPI provides a method which should be used to look up a factory. Web Beans also delegates `Destination` lookup to the container via the SPI.

A.1.6. Resource Services

The resolution of `@Resource` (for injection into simple beans) and the resolution of resources (for injection as a Java EE resource) is delegated to the container. You must provide an implementation of `ResourceServices` which provides these operations. For resolving the `@Resource` injection, Web Beans will provide the `InjectionPoint`; and for Java EE resources, the `jndiName` or `mappedName` will be provided.

A.1.7. Web Services

The resolution of web service references (for injection as a Java EE resource) is delegated to the container. You must provide an implementation of `WebServices` whichprovides this operation. For resolving the Java EE resource, the `jndiName` or `mappedName` will be provided.

A.1.8. The bean store

Web Beans uses a map like structure to store bean instances - `org.jboss.webbeans.context.api.BeanStore`. You may find `org.jboss.webbeans.context.api.helpers.ConcurrentHashMapBeanStore` useful.

A.1.9. Der Applikationskontext

Web Beans expects the Application Server or other container to provide the storage for each application's context. The `org.jboss.webbeans.context.api.BeanStore` should be implemented to provide an application scoped storage.

A.1.10. Bootstrap und Shutdown

Das `org.jboss.webbeans.bootstrap.api.Bootstrap`-Interface definiert den Bootstrap für Web Beans. Um Web Beans zu booten müssen Sie eine Instanz von `org.jboss.webbeans.bootstrap.WebBeansBootstrap` erhalten (die `Boostrap` implementiert), diese über die verwendeten SPIs informieren und dann beim Container um Start anfragen.

The bootstrap is split into phases, bootstrap initialization and boot and shutdown. Initialization will create a manager, and add the standard (specification defined) contexts. Bootstrap will discover EJBs, classes and XML; add beans defined using annotations; add beans defined using XML; and validate all beans.

The bootstrap supports multiple environments. An environment is defined by an implementation of the `Environment` interface. A number of standard envrionments are built in as the enumeration `Environments`. Different environments require different services to be present (for example servlet doesn't require transaction, EJB or JPA services). By default an EE environment is assumed, but you can adjust the environment by calling `bootstrap.setEnvironment()`.

Web Beans uses a generic-typed service registry to allow services to be registered. All services implement the `Service` interface. The service registry allows services to be added and retrieved.

To initialize the bootstrap you call `Bootstrap.initialize()`. Before calling `initialize()`, you must register any services required by your environment. You can do this by calling `bootstrap.getServices().add(JpaServices.class, new MyJpaServices())`. You must also provide the application context bean store.

Nach Aufruf von `initialize()` erhalten Sie den `Manager` durch Aufruf von `Bootstrap.getManager()`.

To boot the container you call `Bootstrap.boot()`.

To shutdown the container you call `Bootstrap.shutdown()` or `webBeansManager.shutdown()`. This allows the container to perform any cleanup operations needed.

A.1.11. JNDI

Web Beans delegates all JNDI operations to the container through the SPI.

Anmerkung

A number of the SPI interface require JNDI lookup, and the class `AbstractResourceServices` provides JNDI/Java EE spec compliant lookup methods.

A.1.12. Laden von Ressourcen

Web Beans needs to load classes and resources from the classpath at various times. By default, they are loaded from the Thread Context ClassLoader if available, if not the same classloader that was used to load Web Beans, however this may not be correct for some environments. If this is case, you can implement `org.jboss.webbeans.spi.ResourceLoader`:

```
    public interface ResourceLoader {

  /**
   * Creates a class from a given FQCN
   *
   * @param name The name of the clsas
   * @return The class
   */
  public Class<?> classForName(String name);

  /**
   * Gets a resource as a URL by name
   *
   * @param name The name of the resource
   * @return An URL to the resource
   */
  public URL getResource(String name);

  /**
   * Gets resources as URLs by name
   *
   * @param name The name of the resource
   * @return An iterable reference to the URLS
   */
  public Iterable<URL
> getResources(String name);

}
```

A.1.13. Servlet injection

Java EE / Servlet does not provide any hooks which can be used to provide injection into Servlets, so Web Beans provides an API to allow the container to request JSR-299 injection for a Servlet.

To be compliant with JSR-299, the container should request servlet injection for each newly instantiated servlet after the constructor returns and before the servlet is placed into service.

To perform injection on a servlet call `WebBeansManager.injectServlet()`. The manager can be obtained from `Bootstrap.getManager()`.

A.2. Der Vertrag mit dem Container

Es gibt eine Reihe von Voraussetzungen, die Web Beans RI dem Container für das korrekte Funktionieren von Implementierungen auferlegen, die außerhalb von Implementierung von APIs fallen.

Klassenlader-Isolierung
: Falls Sie die Web Beans RI in eine Umgebung integrieren, die das Deployment mehrerer Anwendungen unterstützt, so müssen Sie automatisch oder über Benutzerkonfiguration die Klassenlader-Isolierung für jede Web Beans Anwendung aktivieren.

Servlet
: Falls Sie Web Beans in eine Servlet Umgebung integrieren, müssen Sie `org.jboss.webbeans.servlet.WebBeansListener` als einen Servlet-Listener registrieren, entweder automatisch oder aber durch Benutzerkonfiguration. Dies muss für jede Servlet benutzende Web Beans Applikation erfolgen.

JSF
: If you are integrating the Web Beans into a JSF environment you must register `org.jboss.webbeans.jsf.WebBeansPhaseListener` as a phase listener, and `org.jboss.webbeans.el.WebBeansELREsolver` as an EL resolver, either automatically, or through user configuration, for each Web Beans application which uses JSF.

  If you are integrating the Web Beans into a JSF environment you must register `org.jboss.webbeans.servlet.ConversationPropagationFilter` as a Servlet listener, either automatically, or through user configuration, for each Web Beans application which uses JSF. This filter can be registered for all Servlet deployment safely.

Anmerkung

Web Beans only supports JSF 1.2 and above.

Session Bean Interzeptor

Falls Sie Web Beans in eine EJB Umgebung integrieren, müssen Sie `org.jboss.webbeans.ejb.SessionBeanInterceptor` als einen EJB-Interzeptor für alle EJBs in der Applikation registrieren, entweder automatisch oder aber durch Benutzerkonfiguration. Dies muss für jede Enterprise Beans benutzende Web Beans Applikation erfolgen.

Wichtig

You must register the `SessionBeanInterceptor` as the inner most interceptor in the stack for all EJBs.

The `webbeans-core.jar`

If you are integrating the Web Beans into an environment that supports deployment of applications, you must insert the `webbeans-core.jar` into the applications isolated classloader. It cannot be loaded from a shared classloader.

Binding the manager in JNDI

You should bind a `Reference` to the Manager `ObjectFactory` into JNDI at `java:app/Manager`. The type should be `javax.inject.manager.Manager` and the factory class is `org.jboss.webbeans.resources.ManagerObjectFactory`