4

Verteilte Anwendungen

4 Verteilte Anwendungen

Um verschiedene Architekturen im Rahmen einer JEE-Anwendung abbilden zu können müssen zumindest bestimmte Services auch über ein Netzwerkprotokoll angeboten werden können. Genau genommen bedingte ja die Definition des Begriff "Service" genaue diese Verteilbarkeit. Rein "lokale Services" wären durch die Verwendung von Value Objects unnötig kompliziert.

Der JEE-konforme Applikationsserver stellt über die an Hand der Spezifikation geforderten Standard Inbound Connectors (http/https, RMI/IIOP, JMS, SOAP) bereits ein für die meisten Fälle genügendes Repertoire zur Verfügung. Werden weitere Protokolle benötigt so muss ein entsprechender Inbound Connector eventuell programmiert und zusätzlich installiert werden.

4.1 RMI/IIOP

4.1.1 Allgemeines

Stateless und Stateful SessionBeans sind von vorn herein für den Zugriff via RMI ausgelegt. Es soll hier aber nochmals explizit darauf hingewiesen werden, dass dies nicht das eigentliche Kriterium für den EJB-Einsatz ist. Dies sind die Dienste Transaktionssteuerung und Security. Eine Stateless SessionBean, die eine Service-Implementierung um diese Dienste erweitert wird in der Regel stets eine lokale Schnittstelle aufweise, die Remote-Schnittstellen sind optional!

Benötigt beispielsweise das Servlet einer Web-Anwendung einen über Session Facade realisierten Zugriff auf eine Business Object und die Web-Anwendung läuft direkt im Applikationsserver produzieren RMI-Aufrufe nur absolut unnötigen Overhead[1].

[1] Diese Overhead wird häufig durch Optimierungsmaßnahmen der Applikationsserver-Implementierungen kompensiert. Architektonisch sauber ist dies aber nicht.

4.1.2 Exkurs: Session Facade und Business Delegate der J2EE

In der ursprünglichen J2EE-Spezifikation waren Session Beans stark technologisch geprägte Komponenten. Um diese von der Fachlichkeit zu entkoppeln war eine Reihe von Design-Kniffen notwendig, die im Rahmen der J2EE-Patterns gelöst wurden. Diese Patterns sind in dieser Form nicht mehr unbedingt notwendig, werden jedoch auch in der JEE 6 in anderen Ausprägungen benutzt. Deshalb erfolgt hier ein kurzer Ausflug in das Design einer J2EE-Anwendung.

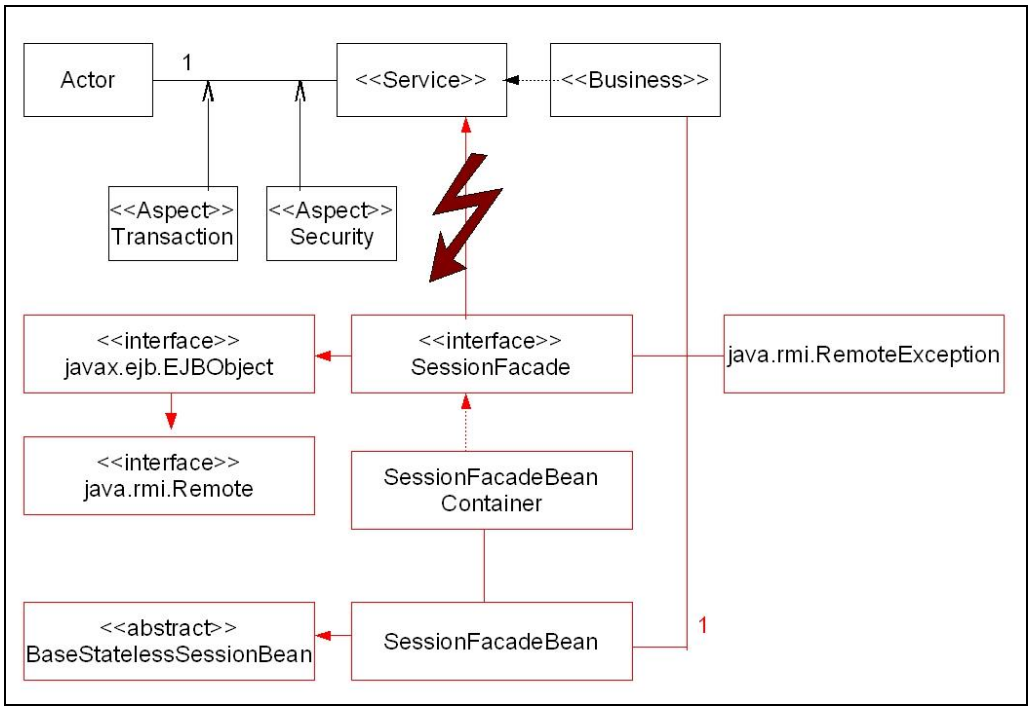
4.1.2.1 Notwendigkeit

Soll der Zugriff auf ein Business Object via RMI ermöglicht werden, so werden Schnittstellen benötigt, die RMI-konform gehalten sind. Dies bedeutet:

- Erweitern der `java.rmi.Remote`-Schnittstelle
- Alle Methoden werfen die `java.rmi.RemoteException`

Diese auf Grund der Java-RMI-Spezifikation notwendigen Bedingungen führen auf den ersten Blick leider zu einer Unschönheit im Design: Die vom Client zu benutzenden Schnittstellen können nicht mehr die Service-Schnittstelle erweitern, da eine abgeleitete Schnittstelle nicht beginnen kann, Methoden der Super-Schnittstelle zusätzliche Exception-Typen werfen zu lassen. Dies ist ja nur für `RuntimeException`s zulässig und die `RemoteException` ist keine. Eine Vererbungshierarchie dieser Form ist somit unzulässig[2]:

[2] Um diese zu ermöglichen müsste die Service-Schnittstelle ja selbst bereits alle Methode mit der `RemoteException` ausgestattet haben. Dies ist aber definitiv keine Plattform-unabhängige Schnittstelle mehr!



Diese Unmöglichkeit hat im Rahmen der JEE-Programmierung zu einigem Aufruhr geführt, noch dazu der direkte Konkurrent, Microsofts .NET diese Problem nicht kennt (dort gibt es eh nur `RuntimeExceptions`). Häufig resultierte dies zu dem Vorwurf, dass das JEE Programmiermodell zu kompliziert sei.

In der Praxis tritt dieses Problem jedoch nur sehr entschärft auf, da ein einfaches Design Pattern eine Lösung liefert. Und das ist der Business Delegate.

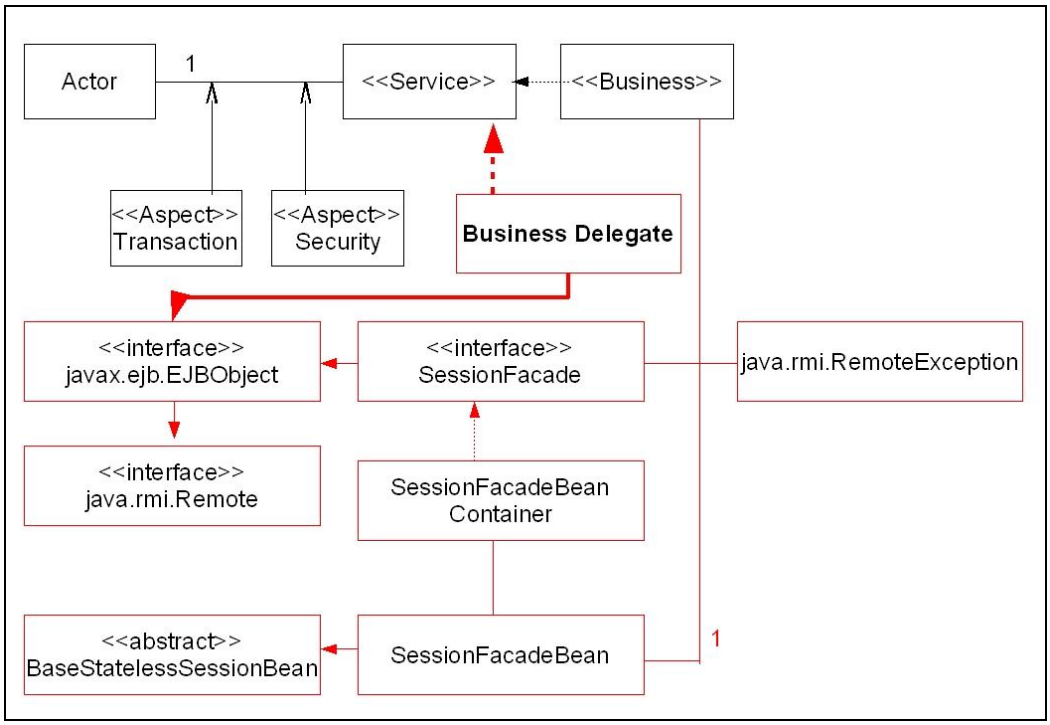
4.1.2.2 Der Business Delegate

Diese neu einzuführende Klasse tritt einfach als Adapter zwischen den `RemoteExceptions` werfenden EJB-Schnittstellen und der Plattformunabhängigen Service-Schnittstelle auf. Allerdings sind seine Aufgaben noch deutlich klarer präzisierbar als eine einfache Adaptions-Logik:

- Der Business Delegate braucht eine Referenz auf die EJB und benutzt dazu den Service Locator.
- Er muss sich um die Behandlung der `RemoteException` kümmern. Die Standard-Implementierung eines Business Delegates wird daraus wohl eine `RuntimeException` machen. Damit verhält sich die JEE-Anwendung genau so wie eine .NET oder auch eine Spring-Anwendung. Das saubere am Business Delegate ist nun jedoch, dass auch komplexere Strategien problemlos konzipiert und eingesetzt werden können:
- Aufruf eines Backup-Systems

- Erzeugen einer Exception, die im Plattform-unabhängigen Modell bereits existiert.
- Retry-Mechanismen
- ...

Der Business Delegate wird auf folgende Art im Klassendiagramm integriert:



Der Business Delegate implementiert die Service-Schnittstelle und delegiert an die zugehörige Session Facade.

Eine Implementierung lautet beispielsweise:

```
public class KeyGeneratorRemoteBusinessDelegate implements
KeyGenerator {


private KeyGeneratorRemoteSessionFacade sessionFacade;


public KeyGeneratorRemoteBusinessDelegate() {
 sessionFacade =
JeeServiceLocator.getStatelessEJBObject(KeyGeneratorRemoteSessi
onFacade.class);
}


public String next() {
 try {
  return sessionFacade.next(1)[0];
```

```
 } catch (RemoteException re) {
 re.printStackTrace();
 throw new RuntimeException(re.getMessage());
 }
}

}
```

Damit dies funktioniert wird der bereits vorhandene Service Locator um die beiden Methoden

```
    public static <T extends EJBHome> T getEJBHome(Class<T>
ejbHomeClass)
            throws ServiceException {
        T result =
ejbHomeClass.cast(ejbHomeCache.get(ejbHomeClass.getName()));
        if (result == null) {
            try {
                InitialContext initialContext = new
InitialContext();
                Field jndiAttribute =
ejbHomeClass.getField("JNDI_NAME");
                Object object = initialContext.lookup((String)
jndiAttribute
                        .get(null));
                Object home =
PortableRemoteObject.narrow(object, ejbHomeClass);
                result = ejbHomeClass.cast(home);
                ejbHomeCache.put(ejbHomeClass.getName(),
result);
            } catch (Exception e) {
                e.printStackTrace();
                throw new ServiceException(e.getMessage());
            }
        }

        return result;

    }
    public static <T extends EJBObject> T
getStatelessEJBObject(
            Class<T> ejbObjectClass) throws ServiceException {
```

```
        T result =
ejbObjectClass.cast(ejbObjectCache.get(ejbObjectClass
                .getName()));
        if (result == null) {
            try {
                InitialContext initialContext = new
InitialContext();
                String homeClassName = ejbObjectClass.getName()
+ "Home";
                Class homeClass = Class.forName(homeClassName);
                Field jndiAttribute =
homeClass.getField("JNDI_NAME");
                Object object = initialContext.lookup((String)
jndiAttribute
                        .get(null));
                EJBHome home = (EJBHome)
PortableRemoteObject.narrow(object,
                        homeClass);
                Method create = homeClass.getMethod("create");
                result =
ejbObjectClass.cast(create.invoke(home));
                ejbObjectCache.put(ejbObjectClass.getName(),
result);
            } catch (Exception e) {
                e.printStackTrace();
                throw new ServiceException(e.getMessage());
            }
        }
        return result;

    }
```

erweitert. Der einzige wesentliche Unterschied ist hier die Verwendung des `javax.rmi.PortableRemoteObjects`, das das CORBA-Narrowing übernimmt.

Die Business Delegate-Klasse kann dann jederzeit von der Service-Factory erzeugt und als Service-Implementierung zurückgegeben werden.

```
public KeyGenerator createBusinessDelegate(){
 return new KeyGeneratorRemoteBusinessDelegate();
}
```

4.1.2.3 Paketstruktur

Die Einführung der zusätzlichen Schicht für die Verteilung über RMI erfordert eine Erweiterung der Paketstruktur:

- <domain>.<service>
- <domain>.<service>.value
- <domain>.<service>.test
- <domain>.<service>.business
- <domain>.<service>.aspects
- <domain>.<service>.ejb
- **<domain>.<service>.network.ejb**
  - Enthält die generierten Schnittstellen für den RMI-Zugriff sowie den Business Delegate

4.1.2.4 Ein generischer Business Delegate

Der Business Delegate übernimmt hier zwar elegant die Adaption zwischen Service und EJB ist aber immer noch eine Klasse, die entweder selbst geschrieben oder pro Session Facade mit Remote-Zugriff generiert werden muss. Der Vorwurf "zu kompliziert" ist damit immer noch gegeben. Allerdings kann diese Aufgabe auch eine einzige Hilfsklasse übernehmen, die den seit Java 1.3 bekannten Dynamic Proxy benutzt:

```
public class GenericBusinessDelegate<T extends EJBObject>
implements InvocationHandler {


private Object delegate;


private Class<T> clazz;


private GenericBusinessDelegate() {
}


protected T getDelegate() {
 if (delegate == null) {
  delegate = JeeServiceLocator.getStatelessEJBObject(clazz);
 }
 return this.clazz.cast(delegate);
}


 public Object invoke(Object proxy, Method toInvoke, Object[]
params)
```

```
 throws Throwable {
 try {
 String methodName = toInvoke.getName();
 Class[] parameterTypes = toInvoke.getParameterTypes();
 Method ejbMethod = clazz.getMethod(methodName,
parameterTypes);
 synchronized (getDelegate()) {
 return ejbMethod.invoke(getDelegate(), params);
 }
 } catch (InvocationTargetException ite) {
 throw ite.getTargetException();
 }
 }


 public static <T extends EJBObject, U> U createProxy(Class<T>
ejbObjectClass,
 Class<U> serviceClass) {
 GenericBusinessDelegate<T> genericServiceBusinessDelegate =
new GenericBusinessDelegate<T>();
 genericServiceBusinessDelegate.setClazz(ejbObjectClass);
 return serviceClass.cast(Proxy.newProxyInstance(
 GenericBusinessDelegate.class.getClassLoader(),
 new Class[] { serviceClass },
genericServiceBusinessDelegate));
 }


 @SuppressWarnings("unchecked")
 public static <T extends EJBObject, U> U createProxy(Class<U>
serviceClass) {
 String packageName = ClassUtils.getPackageName(serviceClass);
 String className = ClassUtils.getShortClassName(serviceClass);
 String ejbObjectClassName = packageName + ".network.ejb." +
className + "RemoteSessionFacade";
 Class<T> ejbObjectClass;
 try {
 ejbObjectClass = (Class<T>) Class.forName(ejbObjectClassName);
 } catch (ClassNotFoundException e) {
 e.printStackTrace();
 throw new RuntimeException(e.getMessage());
 }
```

```
 GenericBusinessDelegate<T> genericServiceBusinessDelegate =
new GenericBusinessDelegate<T>();
 genericServiceBusinessDelegate.setClazz(ejbObjectClass);
 return serviceClass.cast(Proxy.newProxyInstance(
  GenericBusinessDelegate.class.getClassLoader(),
  new Class[] { serviceClass },
genericServiceBusinessDelegate));
 }
 public Class<T> getClazz() {
 return clazz;
 }

 public void setClazz(Class<T> clazz) {
 this.clazz = clazz;
 }
}
```

Diese Klasse verwendet wieder Namenskonventionen, insbesondere wird die im vorherigen Abschnitt eingeführte Paketstruktur.

Die Verwendung des generischen Business Delegates in der Factory benutzt dessen statische Methode:

```
 public KeyGenerator createGenericBusinessDelegate(){
  return
GenericBusinessDelegate.createProxy(KeyGenerator.class);
 }
```

4.1.3 EJB 3

Die Idee des generischen Business Delegates wurde auch bei der Konzeption der EJB 3 mit eingebracht. Diese Klasse sowie die Session Facade wurden in das JEE Framework integriert, das heißt: der JNDI-Lookup liefert hier direkt eine Implementierung der Service-Schnittstelle! Das übernimmt aber eine einfache Erweiterung des Service Locators:

```
public static <T> T getStatelesEJB(Class<T> ejbClass) {
 Object ejbObject;
 try {
  ejbObject = context.lookup(mappings.get(ejbClass.getName()));
 } catch (NamingException e) {
  e.printStackTrace();
  throw new RuntimeException(e.getMessage());
 }
 return cast(ejbObject, ejbClass);
}
```

4.1.4 Remote Facade, spezielle Business Delegates und Transfer Objects

Die eben eingeführten Hilfsklassen bzw. die EJB 3-Spezifikation erleichterten die Einführung einer Remoting-Schicht für RMI gewaltig. Im Endeffekt genügt in beiden Fällen die Einführung einer neuen Klasse, der Bean-Implementierung. Diese wird entweder mit XDoclet- oder EJB 3-Annotations ausgestattet. Die eigentliche Implementierung ist das Setzen der Dependencies und ein triviales Delegieren an das eigentliche Business Object[3].

Es stellt sich nun jedoch recht schnell die Frage, ob diese Trivial-Implementierung in allen Fällen genügt. Anders formuliert lautet die Frage: Wie gut ist die Service-Schnittstelle an die Besonderheiten des RMI-Protokolls in allen Anwendungssituationen angepasst?

Diese Frage mag nun etwas verblüffend erscheinen, da Service-Schnittstellen ja explizit die Forderung der Verteilbarkeit erfüllen müssen und damit sollte die Frage "eigentlich" überflüssig erscheinen.

[3] Prinzipiell könnte bei EJB 3 das Business Object selber bereits die Annotations für die Session Facade enthalten. Diese sind jedoch (leider) teilweise im javax.ejb-Paket gelandet, so dass damit die Forderung der Plattform-unabhängigen Implementierung verletzt wird. Bei einer sauberen Modellierung verlangt also auch EJB 3 die Einführung einer eigenen Klasse.

Eine Betrachtung selbst der einfachsten Service-Schnittstelle ermöglicht nun aber bereits die Konstruktion von Szenarien, in denen beispielsweise die Signatur

```
public String next()
```

"verbessert" werden kann.

Das erste Szenarium fokussiert auf eine extrem schlechte Netzwerkverbindung, in der jedes eingesparte zu übertragende Byte eine Verbesserung bedingt. Ist bekannt, dass der erzeugte Schlüssel aus ASCII-Zeichen besteht, so ist eine Übertragung von Unicode-Zeichen, in denen das höherwertige Byte stets Null ist, überflüssig:

```
public byte[] next()
```

Falls der Schlüssel intern durch eine Zahl repräsentiert wird wäre die nächste Signatur noch "besser" geeignet:

```
public int next()
```

Das zweite Szenarium ist ein anderer Service, der für seine internen Sequenzen eine gewisse Menge an Schlüsseln anfordern muss[4].

[4] Beispielsweise das Einfügen eines komplexen Objektbaums in eine Datenbank mit einem neuen Master-Eintrag und vielen Detail-Einträgen.

So lange der Aufruf des `KeyGenerator`-Services lokal erfolgt ist die folgende Schleife vollkommen unkritisch:

```
for (int i =0; i < 100; i++){
 String key = keygenerator.next();
//...
}
```

Wird nun aber auf Grund einer Architektur-Änderung der Remote-Zugriff notwendig benötigt die obige Schleife ausschließlich auf Grund der Netzwerkverbindung etwa 100 Millisekunden zusätzlich[5]. Nun ist eine Signatur der Form

```
public String[] next(int blockSize);
```

wesentlich performanter zu benutzen.

Alle diese neuen Signaturen sind ganz spezielle Optimierungsmaßnahmen, die natürlich nur in ganz speziellen Situationen hilfreich oder möglich sind. Deshalb ist eine Ergänzung der Service-Schnittstelle um diese Methoden nicht immer richtig bzw. erzeugt unerwünschte Nebeneffekte:

- Alle Business Objekte müssen die neuen Signaturen unterstützen, selbst wenn sie nur in unkritischen Bereichen der Anwendung eingesetzt werden.
- Der Programmierer des Service-Benutzers sieht innerhalb der Service-Schnittstelle Methoden, von denen er nicht unbedingt wissen kann, welche für ihn nun die "richtige" ist. Dokumentationsaufwand und Fehleranfälligkeit des Services steigen damit ungemein.

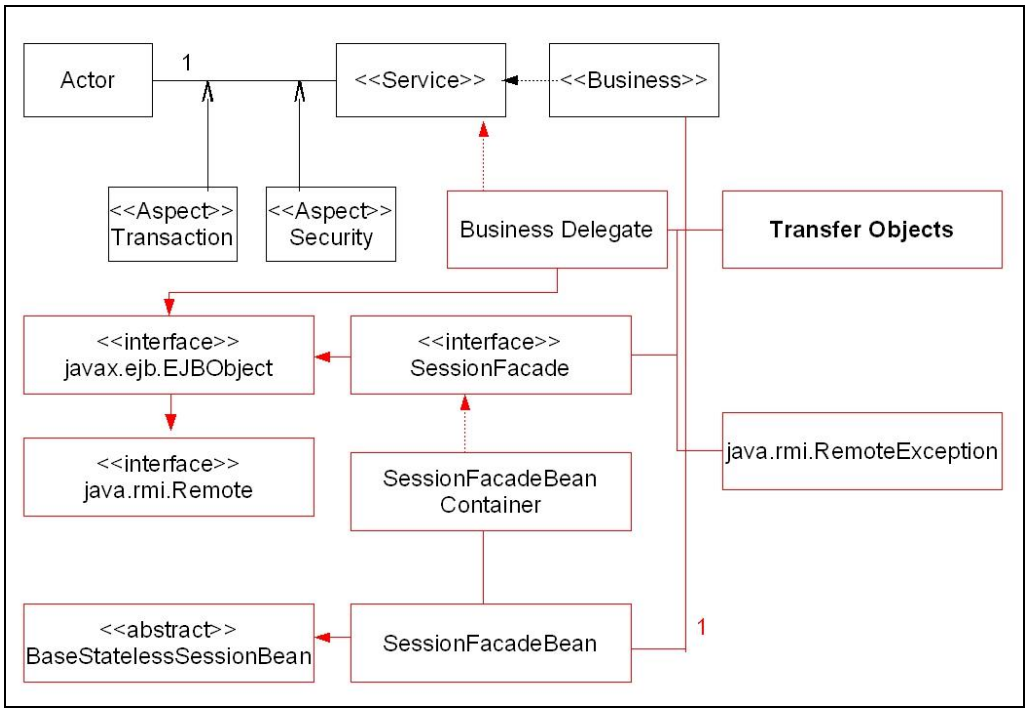
Solche Anpassungen können aber völlig problemlos in das vorhandene Design integriert werden ohne die Service-Schnittstellen ändern zu müssen. Es spricht nichts dagegen, diese Signaturen, die ja ausschließlich in bestimmten Netzwerk-Architekturen sinnvoll sind, dadurch zu realisieren, dass nur die Session Facade diese Methoden anbietet[6]!

Damit ist die Facaden-Logik nicht mehr trivial, sondern sie passt die vom Service gelieferten Value Objekte um in Objekte, die "besser" von einer Schicht der Anwendung in die nächste übertragen werden können: Die (Data) Transfer Objects.

---

[5] Ein RMI-Methodenaufruf benötigt bei typischen Firmen-internen Netzwerken etwa eine Millisekunde.

[6] Diese Argumentation ist natürlich nicht immer stichhaltig bzw. vernünftig. Selbstverständlich gibt es auch Situationen, in denen der Client ganz gezielt bestimmte neue Signaturen benutzen muss. Dann ist es an der Zeit entweder den Service durch weitere Signaturen zu ergänzen oder aber einen neuen Service zu konzipieren, der den ursprünglichen erweitert.



Im folgenden Beispiel wird die Übertragung eines ganzen Blocks von Schlüsseln realisiert. Das Transfer Object ist in diesem Falle ein String-Array.

Der Business Delegate:

```
public class KeyGeneratorBusinessDelegate implements
KeyGenerator {


private static final int CACHE_SIZE;


static {
 CACHE_SIZE = 10000;
}


private String[] cache = new String[0];


private int actualCounter;


{
 actualCounter = CACHE_SIZE;
}


private KeyGeneratorRemoteSessionFacade sessionFacade;
```

```
{
 sessionFacade = JeeServiceLocator

.getStatelessEJBObject(KeyGeneratorRemoteSessionFacade.class);
}


public String next() {
 if (actualCounter == CACHE_SIZE) {
  try {
   System.out.println("Filling cache...");
   cache = sessionFacade.next(CACHE_SIZE);
   actualCounter = 0;
  } catch (RemoteException e) {
   e.printStackTrace();
   throw new RuntimeException(e.getMessage());
  }
 }
 String key = cache[actualCounter];
 actualCounter++;
 return key;
}


}
```

Die Session Facade:

```
public class KeyGeneratorRemoteSessionFacadeBean extends
BaseStatelessSessionBean
 {


private KeyGenerator keyGenerator;


@Override
protected void initSessionBean() {
 keyGenerator =
JeeServiceLocator.getStatelessEJBLocalObject(KeyGeneratorSessio
nFacadeLocal.class);
}
/**
 * Optimierungsmaßnahme für den RMI-Zugriff
 * @ejb.interface-method view-type="remote"
```

```
 * @ejb.transaction type="Required"
 */
public String[] next(int size) {
 String[] keys = new String[size];
 for (int i = 0; i < size; i++) {
  keys[i] = keyGenerator.next();
 }
 return keys;
}
}
```

4.1.5 Erweiterungen

Die folgenden Erweiterungen sind für komplexere Anwendungen zu empfehlen.

4.1.5.1 Local und Remote Session Facade

Nachdem konzeptuell jeder Client, der eine besondere Anpassung der Service-Signatur benötigt, eine eigene spezielle Facade-Signatur benötigt wird die Facade-Implementierung bei größerer Anzahl von Clients naturgemäß immer mächtiger und unübersichtlicher. Dann ist die logisch bereits vorhandene Trennung der Aufgabenverteilung in zwei verschiedene SessionBeans zu erwägen. Die **Local Session Facade** enthält die der Bean zugeordnete fachliche Logik:

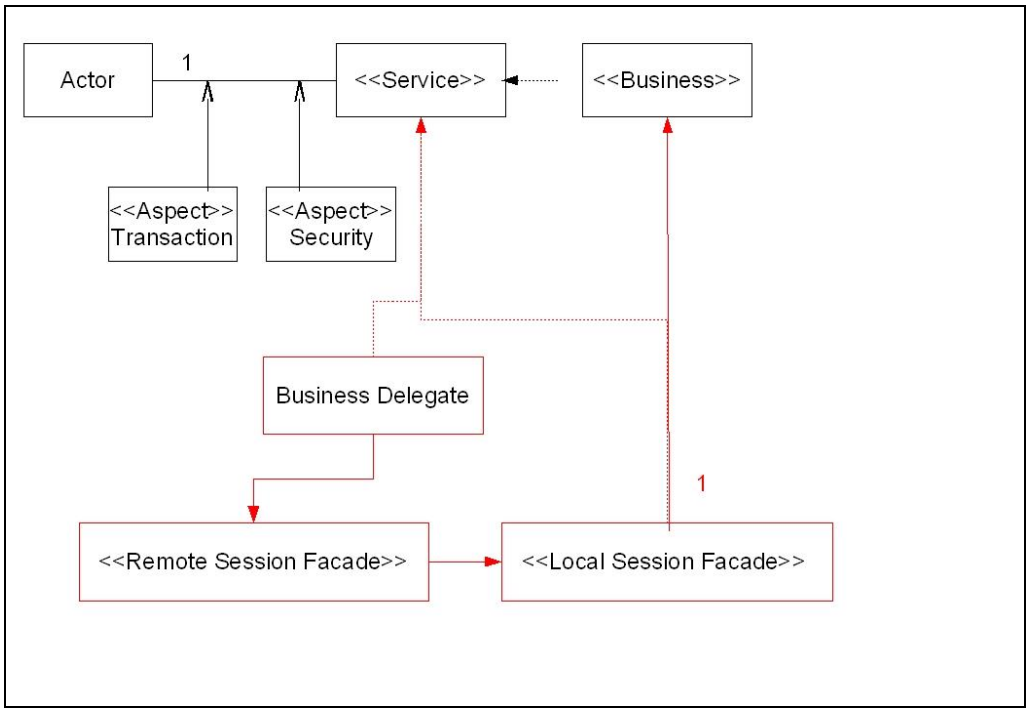
- Transaktionssteuerung
- Security
- Analysieren und Setzen der Dependencies des Business Objects

Eine oder mehrere Remote Session Facaden passen die Signaturen an bzw. erzeugen aus dem von der Business Implementierung gelieferten Value Objects die angepassten Transfer Objects.

Um keine Redundanzen entstehen zu lassen wird die Remote Session Facade an die Local Session Facade delegieren.

Um die Klassendiagramme etwas übersichtlicher gestalten zu können werden im Folgenden drei neue UML-Stereotype eingeführt:

- `<<Local SessionFacade>>` ist eine Stateless SessionBean mit lokalen Schnittstellen.
- `<<Remote SessionFacade>>` ist eine Stateless SessionBean mit ausschließlich Remote-Schnittstellen.
- `<<Business Delegate>>` ist der spezielle Business Delegate.



4.1.5.2 Value Objects und Transfer Objects

Der Katalog der JEE Patterns trifft keine Unterscheidung zwischen Value und Transfer Objects. Früher wurde ausschließlich Value Object, nun Transfer Object benutzt. Der Unterschied zwischen diesen beiden Pattern tritt auch erst bei den eingeführten Optimierungsmaßnahmen auf, vorher gilt tatsächlich Value Object = Transfer Object.

Sobald jedoch eine Remote Session Facade spezielle Objekttypen liefert ergibt sich eine Trennung:

- Value Objects sind Bestandteil der Service-Definition. Sie definieren eine spezielle Client-Sicht auf einen Datenbestand. Value Objects sind auf Grund dieser Definition nicht notwendig serialisierbar.
- Transfer Objects repräsentieren einen Datenbestand, der zu einem Client über ein Netzwerk transferiert werden muss. Transfer Objekte sind damit in irgendeiner Form in einem serialisierbaren Datenstrom umzuwandeln.

Ein Beispiel für diesen Unterschied liefert der `BooksService`. Die speziellen Client-Sichten `BookValue` und `BookListValue` treten in der Signatur der Service-Schnittstelle auf. Unter anderem gibt es die Methode

```
BookValue updateBook(BookValue bookDetailValue) throws
BookException;
```

Diese Methode sendet geänderte Buch-Informationen zum Server, der dann in einer realen Implementierung die geänderten Werte in einer Datenbank abspeichern wird. Beginnt man aber, diese Sequenzen umzusetzen wird schnell ein Problem erkennbar: Woher "weiß" der Server denn, welche Werte und Attribute überhaupt geändert wurden? Der Client-Anwendung sind diese Informationen bekannt, aber wie kann diese Zusatz-Information sauber transferiert werden[7]? Es ist hier sinnvoll, nur die geänderten Werte zu übertragen. Für flache Value Objekte genügt hierfür eine `Map`.

Damit der Business Delegate erkennen kann, welche Attribute geändert wurden, wird eine Subklasse von `BookValue` geschrieben, die sich Zustandsänderungen merken kann[8]:

```
public class SmartBookValue extends BookValue {

private boolean ready;

public SmartBookValue(BookValue bv) {
 try {
  PropertyUtils.copyProperties(this, bv);
 } catch (Exception e) {
  e.printStackTrace();
  throw new RuntimeException(e.getMessage());
 }
 ready = true;
}

HashMap<String, Object> dirtyValues;

{
 dirtyValues = new HashMap<String, Object>();
}

private static final long serialVersionUID = 1L;

@Override
public void setAuthorNames(List names) {
```

[7] Natürlich könnte das gesendete `BookValue` nur die Attribute belegt bekommen, die geändert wurden. Alle anderen stehen auf symbolischen Werten.

[8] Auf Grund dieser Logik ist dies im strengen Sinn kein Value Object mehr.

```
 if (ready) {
  if (this.getAuthorNames() == null && names != null) {
   dirtyValues.put("authorNames", names);
  }
  if ((this.getAuthorNames() != null)
    && !this.getAuthorNames().equals(names)) {
   dirtyValues.put("authorNames", names);
  }
 }
 super.setAuthorNames(names);
}

@Override
public void setAvailable(boolean b) {
 if (ready) {
  if (this.isAvailable() != b) {
   dirtyValues.put("available", b);
  }
 }
 super.setAvailable(b);
}

@Override
public void setDescription(String description) {
 if (ready) {
  if (this.getDescription() == null && description != null) {
   dirtyValues.put("description", description);
  }
  if (this.getDescription() != null
    && !this.getDescription().equals(description)) {
   dirtyValues.put("description", description);
  }
 }
 super.setDescription(description);
}

@Override
public void setIsbn(String isbn) {
 if (ready) {
```

```
 if (this.getIsbn() == null && isbn != null) {
  dirtyValues.put("isbn", isbn);
 }
 if (this.getIsbn() != null && !this.getIsbn().equals(isbn)) {
  dirtyValues.put("isbn", isbn);
 }
 }
 super.setIsbn(isbn);
}

@Override
public void setPrice(double d) {
 if (ready) {
 if (this.getPrice() != d) {
  dirtyValues.put("price", d);
 }
 }
 super.setPrice(d);
}

@Override
public void setTitle(String title) {
 if (ready) {
 if (this.getTitle() == null && title != null) {
  dirtyValues.put("title", title);
 }

 if (this.getTitle() != null && !this.getTitle().equals(title))
{
  dirtyValues.put("title", title);
 }
 }
 super.setTitle(title);
}

public Map<String, Object> getDirtyValues() {
 return dirtyValues;
}
}
```

Der Business Delegate sendet nur die `Map` mit den "Dirty Values":

```
public class OptimizedBooksServiceBusinessDelegate extends

 BaseStatelessEjbObjectBusinessDelegate<BooksServiceRemoteSessio
nFacade>
 implements BooksService {


private BooksServiceRemoteSessionFacade delegate;


public OptimizedBooksServiceBusinessDelegate() {
 super(BooksServiceRemoteSessionFacade.class);
 delegate = getDelegate();
}


public void deleteBookByIsbn(String isbn) throws BookException
{
 try {
  delegate.deleteBookByIsbn(isbn);
 } catch (RemoteException re) {
  re.printStackTrace();
  throw new RuntimeException(re.getMessage());
 }
}


@SuppressWarnings("unchecked")
public Collection<BookListValue> findAllBooks() throws
BookException {
 try {
  return delegate.findAllBooks();
 } catch (RemoteException re) {
  re.printStackTrace();
  throw new RuntimeException(re.getMessage());
 }
}


public BookValue findBookByIsbn(String isbn) throws
BookException {
 try {
  return new SmartBookValue(delegate.findBookByIsbn(isbn));
 } catch (RemoteException re) {
```

```
  re.printStackTrace();
  throw new RuntimeException(re.getMessage());
 }
}

public String newBook(String title) throws BookException {
 try {
 return delegate.newBook(title);
 } catch (RemoteException re) {
 re.printStackTrace();
 throw new RuntimeException(re.getMessage());
 }
}

public BookValue updateBook(BookValue bookDetailValue) throws
BookException {
 try {
 if (bookDetailValue instanceof SmartBookValue) {

  SmartBookValue sbv = (SmartBookValue) bookDetailValue;
  System.out.println("Updating Book using dirtyValues " +
sbv.dirtyValues);
  return new SmartBookValue(delegate.updateBook(sbv.getIsbn(),
   sbv.dirtyValues));
 } else {
  return delegate.updateBook(bookDetailValue);
 }
 } catch (RemoteException re) {
 re.printStackTrace();
 throw new RuntimeException(re.getMessage());
 }
}

}
```

Die spezielle Remote Session Facade empfängt und schreibt ausschließlich die geänderten Werte:

```
public class BooksServiceSessionFacadeBean extends
BaseStatelessSessionBean
 implements BooksService {


private MapBooksService booksService;


@Override
protected void initSessionBean() {
 booksService = new MapBooksService();
 RandomKeyGenerator generator = new RandomKeyGenerator();
 booksService.setKeyGenerator(generator);
 //
booksService.setKeyGenerator(GenericEJBObjectBusinessDelegate.c
reateProxy(KeyGenerator.class));
 //
booksService.setKeyGenerator(ServiceLocator.getStatelessEJBLoca
lObject(KeyGeneratorSessionFacadeLocal.class));
 /*
  * //TOTAL FALSCH!!!! KeyGeneratorSessionFacadeBean bean = new
  * KeyGeneratorSessionFacadeBean();
  * bean.setSessionContext(this.getSessionContext());
bean.ejbCreate();
  * booksService.setKeyGenerator(bean);
  */
 StoreServiceMock mock = new StoreServiceMock();
 booksService.setStoreService(mock);
}


@Override
protected void destroySessionBean() {
 // TODO Auto-generated method stub


}


/**
 * @ejb.interface-method
 * @ejb.transaction type="Required"
 */
public void deleteBookByIsbn(String isbn) throws BookException
{
 booksService.deleteBookByIsbn(isbn);
```

```
}

/**
 * @ejb.interface-method
 * @ejb.transaction type="Required"
 */
public Collection<BookListValue> findAllBooks() throws
BookException {
 return booksService.findAllBooks();
}

/**
 * @ejb.interface-method
 * @ejb.transaction type="Required"
 */
public BookValue findBookByIsbn(String isbn) throws
BookException {
 return booksService.findBookByIsbn(isbn);
}

/**
 * @ejb.interface-method
 * @ejb.transaction type="Required"
 */
public String newBook(String title) throws BookException {
 return booksService.newBook(title);
}

/**
 * @ejb.interface-method
 * @ejb.transaction type="Required"
 */
public BookValue updateBook(BookValue bookDetailValue) throws
BookException {
 return booksService.updateBook(bookDetailValue);
}

/**
 * @ejb.interface-method
 * @ejb.transaction type="Required"
 */
```

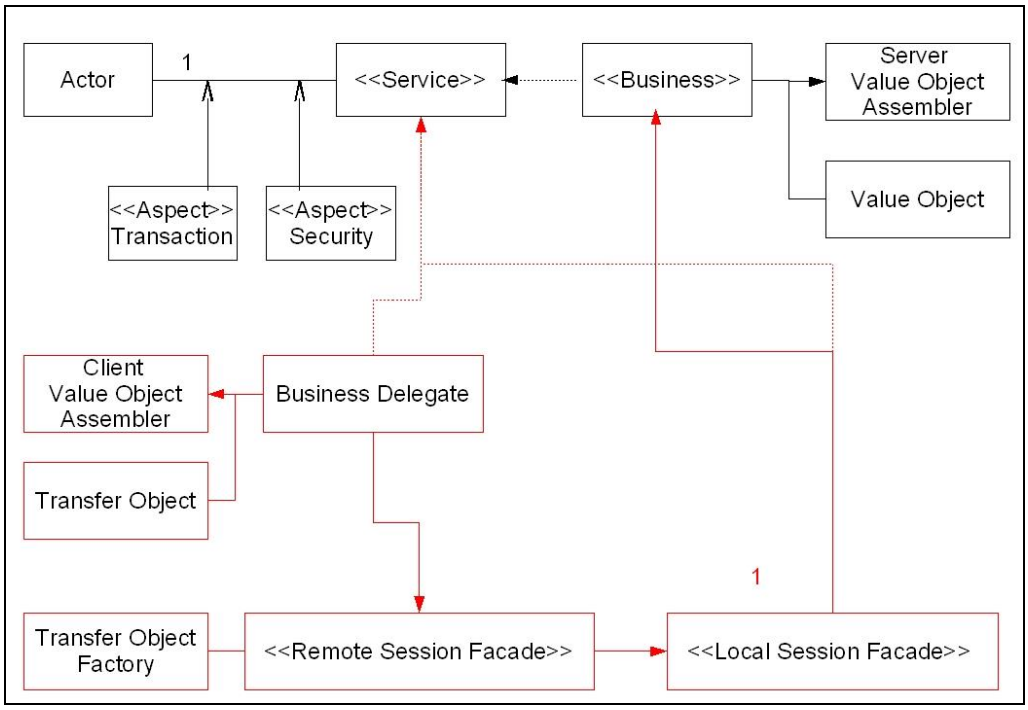
```
public BookValue updateBook(String isbn, Map<String, Object>
dirtyValues)
 throws BookException {
BookValue bv = findBookByIsbn(isbn);
System.out.println("Merging dirtyValues " + dirtyValues + "
with bookValue " + bv);
for (String key : dirtyValues.keySet()) {
Object value = dirtyValues.get(key);
try {
 PropertyUtils.setProperty(bv, key, value);
} catch (Exception e) {
 throw new RuntimeException(e.getMessage());
}
}
return booksService.updateBook(bv);
}

}
```

4.1.5.3 Value Object Assembler und Data Transfer Object Factory

Diese beiden Pattern sind ebenfalls Bestandteil des JEE Katalogs, wenn auch in jeweils unterschiedlichen Versionen. Der ursprüngliche Value Object Assembler diente dazu, aus Server-seitigen Objekten Objekte zu erzeugen, die zum Client gesendet werden können. Diese Rolle übernimmt nun die Data Transfer Object Factory.

Die ursprüngliche Aufgabe ist nun dem Business Object zugeordnet. Dieses kennt und benutzt in komplexeren Anwendungen mehrere Services, um das benötigte Value Object zu erzeugen bzw. eben "aufzusammeln". Auch hier enthält die Buchanwendung ein Beispiel: Das `available`-Attribut des `BookValue` wird vom `StockService` bereitgestellt.

Der Value Object Assembler wird nun jedoch auch auf Client-Seite benötigt. Hier muss der Business Delegate aus dem Transfer Object die Value Objects erzeugen.



Diese Entwurfsmuster werden in einfachen Szenarien nicht durch eigene Klassen realisiert. Sie genügen dem typische Refactoring-Vorgang Sequenz -> private Methode -> Hilfsklasse.

4.1.5.4 Value und Transfer Documents

Die Entwurfsmuster Value Object bzw. Transfer Object fixieren auf eine Objektorientierte Modellierung der Schnittstellen mit typisierten Objekten. Dies ist nicht immer richtig: Im Rahmen einer Dokumentenorientierten Programmierung wird statt eines Typs ein Dokument verwendet, das einem, falls erwünscht validierbarem Schema genügt. Die Signatur lautet dann:

```
public Document execute (Document document);
```

`Document` kann, muss aber nicht ein XML-Dokument sein. Auch eine Zeichenkette, ein `javax.servlet.HttpServletRequest` oder eine `javax.jms.Message` sind geeignete Dokument-Typen.

In solchen Fällen spricht man dann natürlich sinnvoller von Value Document bzw. Transfer Document.

4.1.5.5 Value List Handler

Eine weitere typische Optimierungsmaßnahme ist die Einführung einer aus der Datenbank-Programmierung bekannten Fetch Size für RMI-Methoden. Bei einer Datenbank-Abfrage mit potenziell vielen Treffern enthält das gelieferte `ResultSet` nicht bereits die komplette Ergebnismenge, sondern einen Teilausschnitt sowie einen Cursor auf die innerhalb der Datenbank gehaltenen Treffermenge.

So ein Verhalten kann auch bei Service-Schnittstellen sinnvoll sein. Dann spricht man vom Design Pattern Value List Handler. Dem `ResultSet` entspricht hierbei der Rückgabetyp `java.util.Collection` oder eine ihrer Sub-Schnittstellen.

Ist die Einführung eines Value List Handlers bereits bei der Konzeption der Service-Schnittstelle klar, so könnten die kritischen Suchmethoden auf folgende Art und Weise geschrieben werden:

```
public int prepareSearch ();
public Collection next(int fetchSize);
public void discard();
```

`prepareSearch` stößt auf Server-Seite die Suche an und liefert die Anzahl der Treffer, `next` liefert dann jeweils den nächsten Block von Ergebnissen. Dieser Ansatz ist jedoch kritisch: Der eben definierte Service ist nicht mehr Zustandslos, da die Ergebnismenge auf dem Server irgendwie gehalten werden muss. Dies erfolgt entweder dadurch, dass der Server pro Client-Anfrage die gesamte Treffermenge in seinen Hauptspeicher einlädt oder aber das zugehörige `ResultSet` bleibt zwischen den einzelnen Aufrufen geöffnet. So eine Service-Schnittstelle

kann deshalb in dieser Form nicht Bestandteil eines Platform Independent Models sein.

Die nächste Service-Schnittstelle ist Zustandslos:

```
public int count ();
public Collection next(int start, int fetchSize);
public void discard();
```

Allerdings ist die Implementierung der `next`-Methode nur extrem aufwändig zu realisieren. Mit hoher Wahrscheinlichkeit wird diese "Optimierung" dazu führen, dass die Datenbank die Abfrage mit vielen Treffern nun sogar mehrfach komplett ausführen muss.

Die Ansätze auf der Ebene der Service-Schnittstelle sind somit alle nicht sonderlich befriedigend. Verlagert man das Problem in das Platform Specific Model für die JEE so ist die erste Alternative durch die Einführung einer Stateful SessionBean durchaus machbar. Nachdem der Service aber weiterhin seine Zustandslose Formulierung bekommen kann wird in diesem Beispiel der Value List Handler wiederum durch das Zusammenspiel Business Delegate – Session Facade realisiert.

Die zugehörige Remote Session Facade[9] ist nun Zustandsbehaftet:

```
public class ValueListHandlerFacadeBean extends
BaseStatefulSessionBean {


private BooksService booksService;
private List<BookListValue> allBooks;
private int deliveredBooks;
@Override
protected void initSessionBean() {
 booksService =
JeeServiceLocator.getStatelessEJBLocalObject(BooksServiceSessio
nFacadeLocal.class);
}


@Override
/**
 * @ejb.create-method
 *
 */
public void ejbCreate(){
```

[9] Für lokale Zugriffe ist der Value List Handler unsinnig.

```
}

/**
 *@ejb.interface-method
 * @return
 */
public int findAllBooks() throws BookException{
 allBooks = new
ArrayList<BookListValue>(booksService.findAllBooks());
 deliveredBooks = 0;
 return allBooks.size();
}

/**
 * @ejb.interface-method
 * @param fetch
 * @return
 */
public Collection<BookListValue> getNext(int fetch){
 int endSize = deliveredBooks + fetch;
 List<BookListValue> result = allBooks.subList(deliveredBooks,
endSize);
 deliveredBooks = endSize;
 return new ArrayList<BookListValue>(result);
}
}
```

Nachdem die Service-Schnittstelle dem Client keine Iterationslogik anbieten kann liefert der Business Delegate eine spezielle Collection-Implementierung. Diese hält sich die geladenen Daten in einer internen Liste und weiß, wie Daten nachgeladen werden können:

```
public class ValueListHandlerCollection extends
AbstractList<BookListValue> {


private int size;

private ArrayList<BookListValue> loadedElements;

private ValueListHandlerBusinessDelegate
valueListHandlerBusinessDelegate;


{
 loadedElements = new ArrayList<BookListValue>();
}


@Override
public BookListValue get(int index) {
 int loadedSize = loadedElements.size();
 if (loadedSize <= index){

loadedElements.addAll(valueListHandlerBusinessDelegate.getNext(
(index - loadedSize + 1)));
 }
 if (loadedElements.size() == size){
  valueListHandlerBusinessDelegate.remove();
 }
 return loadedElements.get(index);
}


@Override
public int size() {
 return getSize();
}


public int getSize() {
 return size;
}


public void setSize(int size) {
```

```
 this.size = size;
}


public ValueListHandlerBusinessDelegate
getValueListHandlerBusinessDelegate() {
 return valueListHandlerBusinessDelegate;
}


public void setValueListHandlerBusinessDelegate(
 ValueListHandlerBusinessDelegate
valueListHandlerBusinessDelegate) {
 this.valueListHandlerBusinessDelegate =
valueListHandlerBusinessDelegate;
}


}
```

Der Business Delegate erzeugt nun bei der entsprechenden Suche eine Instanz dieser Collection und setzt sich als Attribut, um der Collection das Nachladen zu ermöglichen:

```
public class ValueListHandlerBusinessDelegate implements
BooksService {
private BooksService booksService;
private boolean removed;
private ValueListHandlerFacade valueListHandlerFacade;


{
booksService = GenericBusinessDelegate
 .createProxy(BooksService.class);
try {
 valueListHandlerFacade = JeeServiceLocator.getEJBHome(
  ValueListHandlerFacadeHome.class).create();
} catch (Exception e) {
 e.printStackTrace();
 throw new RuntimeException(e.getMessage());
}
Runtime.getRuntime().addShutdownHook(new Thread(){
 public void run(){
  if(! removed){
   remove();
```

```
  }
 }
});
}

public void deleteBookByIsbn(String isbn) throws BookException
{
booksService.deleteBookByIsbn(isbn);
}

public Collection<BookListValue> findAllBooks() throws
BookException {
ValueListHandlerCollection valueListHandlerCollection = new
ValueListHandlerCollection();
try {

valueListHandlerCollection.setSize(valueListHandlerFacade.findA
llBooks());

valueListHandlerCollection.setValueListHandlerBusinessDelegate(
this);
 return valueListHandlerCollection;
} catch (RemoteException e) {
 e.printStackTrace();
 throw new RuntimeException(e.getMessage());
}
}

public BookValue findBookByIsbn(String isbn) throws
BookException {
return booksService.findBookByIsbn(isbn);
}

public String newBook(String title) throws BookException {
return booksService.newBook(title);
}

public BookValue updateBook(BookValue bookDetailValue) throws
BookException {
return booksService.updateBook(bookDetailValue);
}
```

```
@SuppressWarnings("unchecked") Collection<BookListValue>
getNext(int fetch){
 try {
  return valueListHandlerFacade.getNext(fetch);
 } catch (RemoteException e) {
  e.printStackTrace();
  throw new RuntimeException(e.getMessage());
 }
}

void remove(){
 removed = true;
 try {
  valueListHandlerFacade.remove();
 } catch (Exception e) {
  e.printStackTrace();
 }
}
}
```

Der Client bemerkt von der Existenz des Value List Handlers nichts: Er bekommt in der ursprünglichen Form eine vollständig gefüllte Collection, danach eine Collection mit Nachlade-Logik.

Die Optimierung mit Hilfe des Value List Handlers funktioniert nur mit einigen Nebenbedingungen:

- Der Server kann Client-behafteten Zustand im Hauptspeicher halten. Dies ist in der JEE bei Stateful SessionBeans und Servlets der Fall.
- Das eigentliche Laden der großen Datenmengen von der Datenbank ist nicht das eigentliche Problem, sondern die Übertragung zu einem entfernten Client. Liegt das Performance-Problem in der Datenbank so bringt die Einführung eines Value List Handlers kaum Vorteile.
- Die Sequenzen des Clients können bereits mit der Nicht-vollständig gefüllten Collection arbeiten. Dies ist besonders klar bei Clients mit einer Benutzerschnittstelle (GUI-Anwendung oder Web Frontend). Soll beispielsweise eine Swing-Tabelle mit den Ergebnissen befüllt werden, so genügt dem Tabellenmodell die Information „Anzahl der Treffer“ sowie die ersten in der Tabelle dargestellten Daten. Völlig ungeeignet ist der Value List Handler für eine Verarbeitung, die auf der gesamten Treffermenge arbeiten muss, z.B. Reporting. Hier führt die Einführung dieses Patterns nur zu einer Erhöhung der Anzahl der Remote Methodenaufrufe und ist damit Performance-mindernd.

4.2 Die Message Facade

4.2.1 Vorteile des JMS-Protokolls

Der Zugriff auf die Local Session Facade über eine vorgeschaltete Remote Session Facade ist nicht immer notwendig. Alternativ oder ergänzend kann statt des RMI-Protokolls auch JMS verwendet werden. Die Entscheidung darüber ist Bestandteil der Anwendungs-Architektur. Kriterien dafür lauten:

- Ausfallsicherheit und Nachvollziehbarkeit: Hier ist ein JMS-System auf Grund des Message Puffers der JMS-Destinations häufig die geeignete Wahl.
- Flexibilität: Die Session Facaden werden über typisierte entfernte Referenzen angesprochen. Ein flexibles Ergänzen durch weitere Funktionalitäten verlangt relativ viel Programmieraufwand[10].
- JMS-Provider bieten häufig APIs für verschiedene Programmiersprachen. Deshalb kann Messaging auch für die Interoperabilität eingesetzt werden.
- Asynchrone Aufrufe oder Rückgaben via Callback-Aufrufen sind integraler Bestandteil von JMS.

[10] In einem späteren Abschnitt wird diese Flexibilität jedoch auch für Session Facaden eingeführt.

4.2.2 Layer Supertype und Basisklasse für den Business Delegate

Die bisher vorhandenen Service-Definitionen sind Schnittstellenorientiert ausgelegt worden. Um die Umstellung des Netzwerkprotokolls auf JMS zu ermöglichen müssen spezielle Klassen erzeugt werden:

- Auf Client-Seite muss ein spezieller Business Delegate alle Methodenaufrufe in eine Message umwandeln und zum Server senden. Auf einer speziellen Destination wartet der Business Delegate dann auf die Antwort-Message des Servers und extrahiert daraus das Ergebnis. Dies ist entweder das normale Resultat oder eine Exception.
- Auf Server-seite muss eine `javax.jms.MessageListener` oder eine `javax.ejb.MessageDrivenBean` die empfangene Message analysieren, den Aufruf der Session Facade durchführen und das Ergebnis zum Client zurücksenden.

Diese Aufgaben sind relativ komplex, können jedoch in geeignete Hilfsklassen ausgelagert werden:

Die abstrakte Basisklasse für Message Driven Beans übernimmt die Weiterleitung von Messages an entsprechen typisierte `process<Typ>`-Methoden:

```
public abstract class BaseMessageDrivenBean implements
MessageDrivenBean,
 MessageListener {
private MessageDrivenContext messageDrivenContext;


public void setMessageDrivenContext(
 MessageDrivenContext messageDrivenContext) throws EJBException
{
 this.messageDrivenContext = messageDrivenContext;
 initMessageDrivenBean();
}


public void ejbCreate() {
}


public void initMessageDrivenBean(){}


public void destroyMessageDrivenBean(){}


public void ejbRemove() throws EJBException {
 destroyMessageDrivenBean();
}
```

```
public void onMessage(Message message) {
 try {
 if (message instanceof TextMessage) {
  processTextMessage((TextMessage) message);
 } else if (message instanceof ObjectMessage) {
  processObjectMessage((ObjectMessage) message);
 } else if (message instanceof BytesMessage) {
  processBytesMessage((BytesMessage) message);
 } else if (message instanceof StreamMessage) {
  processStreamMessage((StreamMessage) message);
 } else if (message instanceof MapMessage) {
  processMapMessage((MapMessage) message);
 } else {
  processMessage(message);
 }
 } catch (JMSException e) {
 handleJMSException(e);
 }
}

 protected void processTextMessage(TextMessage message) throws
JMSException {
 throw new UnsupportedOperationException("processMethod must be
overriden");
 }

 protected void processObjectMessage(ObjectMessage message)
  throws JMSException {
 throw new UnsupportedOperationException("processMethod must be
overriden");
 }

 protected void processBytesMessage(BytesMessage message)
  throws JMSException {
 throw new UnsupportedOperationException("processMethod must be
overriden");
 }

 protected void processStreamMessage(StreamMessage message)
```

```
 throws JMSException {
 throw new UnsupportedOperationException("processMethod must be
overriden");
 }

 protected void processMapMessage(MapMessage message) throws
JMSException {
 throw new UnsupportedOperationException("processMethod must be
overriden");
 }

 protected void processMessage(Message message) throws
JMSException {
 throw new UnsupportedOperationException("processMethod must be
overriden");
 }

 public MessageDrivenContext getMessageDrivenContext() {
 return messageDrivenContext;
 }

 protected void handleJMSException(JMSException e){
 e.printStackTrace();
 }
}
```

Um die Methoden-Parameter eines Aufrufs zu bekommen und das Ergebnis eines Aufrufs zum Client zurückgesendet werden kann wird `javax.jms.ObjectMessage` verwendet. Eine weitere, ebenfalls abstrakte Klasse schreibt das Ergebnis eines Aufrufs in eine `ObjectMessage`, die an die in der ursprünglichen Request-Message gesetzten Destination JMSReplyTo gesendet wird:

```
public abstract class BaseRespondingMessageDrivenBean extends
 BaseMessageDrivenBean {

 private QueueConnectionFactory connectionFactory;

 public BaseRespondingMessageDrivenBean(String jndiName){
 connectionFactory = getConnectionFactory(jndiName);
 }
 public abstract ObjectMessage getResponse(Session session,
```

```
 ObjectMessage message) throws JMSException;



public QueueConnectionFactory getConnectionFactory(String
jndiName) {
 try {
  InitialContext context = new InitialContext();
  return (QueueConnectionFactory) context
   .lookup("java:comp/env/ConnectionFactory");
 } catch (Exception e) {
  e.printStackTrace();
  throw new RuntimeException(e.getMessage());
 }

}


protected final void processObjectMessage(ObjectMessage
message) {
 QueueConnection connection = null;
 QueueSession session = null;
 QueueSender messageProducer = null;
 try {
  connection = connectionFactory.createQueueConnection();
  session = connection.createQueueSession(false,
   QueueSession.SESSION_TRANSACTED);
  messageProducer =
session.createSender((Queue)message.getJMSReplyTo());
  messageProducer.send(getResponse(session, message));

 } catch (JMSException e) {
  handleJMSException(e);
 } finally {
  if (messageProducer != null) {
   try {
    messageProducer.close();
   } catch (JMSException e) {
    e.printStackTrace();
   }
  }
  if (session != null) {
```

```
  try {
   session.close();
  } catch (JMSException e) {
   e.printStackTrace();
  }

 }
 if (connection != null) {
  try {
   connection.close();
  } catch (JMSException e) {
   e.printStackTrace();
  }
 }
}
}
```

Soll der `KeyGenerator` über JMS-Protokoll aufgerufen werden, so werden dafür zwei einfache Implementierungen geschrieben:

Die eigentliche Message Facade-Implementierung lautet:

```
public class KeyGeneratorMessageFacadeBean extends
 BaseRespondingMessageDrivenBean {
public KeyGeneratorMessageFacadeBean(){
 super("ConnectionFactory");
}
private static final long serialVersionUID = 1L;
private RandomKeyGenerator randomKeyGenerator;
public ObjectMessage getResponse(Session session, ObjectMessage
message)
 throws JMSException {
System.out.println("Message Received with command: " +
message.getStringProperty("command"));
 ObjectMessage response = session.createObjectMessage();
 response.setObject(new
Result<String>(randomKeyGenerator.next()));
 return response;
}

public void initMessageDrivenBean() {
 randomKeyGenerator = new RandomKeyGenerator();

}

}
```

Der Business Delegate lautet:

```
public class KeyGeneratorJmsBusinessDelegate implements KeyGen-
erator {


public String next() {
 try {
  InitialContext context = new InitialContext();
  ConnectionFactory connectionFactory = (ConnectionFactory) con-
text
   .lookup("ConnectionFactory");
  Connection connection = connectionFactory.createConnection();
  Session session = connection.createSession(false,
   Session.AUTO_ACKNOWLEDGE);
  TemporaryQueue temporaryQueue = ses-
sion.createTemporaryQueue();
  Message requestMessage = session.createMessage();
  requestMessage.setJMSReplyTo(temporaryQueue);
  Destination requestDestination = (Destination) context
   .lookup("queue/A");
  MessageProducer messageProducer = session
   .createProducer(requestDestination);
  messageProducer.send(requestMessage);
  messageProducer.close();
  connection.start();
  MessageConsumer messageConsumer = session
   .createConsumer(temporaryQueue);
  Message responseMessage = messageConsumer.receive();
  String generatedKey = responseMes-
sage.getStringProperty("result");
  if (generatedKey == null) {
   throw new RuntimeException(responseMessage
    .getStringProperty("exception"));
  }
  messageConsumer.close();
  connection.stop();
  temporaryQueue.delete();
  session.close();
  connection.close();


  return generatedKey;
```

```
 } catch (Exception e) {
  e.printStackTrace();
  throw new RuntimeException(e.getMessage());
 }
}

}
```

4.2.3 Der Service to Worker

4.2.3.1 Einführung

Die oben am Beispiel des Schlüsselgenerators demonstrierte Vorgehensweise ist für komplexere Service-Schnittstellen immer noch aufwändig zu programmieren. Der Programmierer muss sich ein geeignetes Protokoll ausdenken und innerhalb der Server-Implementierung müssen die Messages entsprechend analysiert werden.

Andererseits lässt sich die Aufgabenstellung recht allgemein formulieren:

- Über das Netzwerk wird ein Aufruf empfangen. Darin stehen Informationen, die eindeutig eine Service-Methode kennzeichnen. Zusätzlich sind eventuell notwendige Parameter enthalten.
- Die Server-Implementierung analysiert den Request und ruft die Service-Methode auf.
- Das Ergebnis des Aufrufs kann verschiedene Zustände aufweisen ("OK", fachliche Ausnahme, Laufzeitfehler) und wird vom Server in irgendeiner Form als Antwort über das Netz gesendet.

Diese Sequenzen lassen sich sehr schön mit dem Design Pattern „Command“ umsetzen. Allerdings sind eben noch einige Erweiterungen, insbesondere die Berücksichtigung der Netzwerk-Kommunikation notwendig. Deshalb wird dieses Pattern in diesem Zusammenhang als „Service to Worker“ bezeichnet[11].

[11] Service to Worker ist aus der Historie her eigentlich ein JEE Pattern der Präsentationsschicht und wird häufig im Zusammenhang mit Web-Anwendungen benutzt. Dieses Entwurfsmuster lässt sich aber vollkommen problemlos für die oben geschilderte allgemeine Problemstellung nutzen.

4.2.3.2 Das Command-Framework

Das Command-Framework besteht aus drei Hilfsklassen:

- Ein `Result` hat einen internen Zustand und hält ein Ergebnis-Objekt. Ergebnis-Objekte müssen über ein Netzwerk transferiert werden können:

```
public class Result<T> implements Serializable {
 private static final long serialVersionUID = 1L;
 private final ResultState state;
 private T result;
 private Exception exception;
 /**
  * result for returning void method
  */
 public static final Object VOID;
 static {
  VOID = new Serializable() {

   private static final long serialVersionUID = 1L;
  };
 }
 public Result(T result) {
  this.state = ResultState.OK;
  this.result = result;
 }
 public Result(Exception exception) {
  this.state = ResultState.EXCEPTION;
  this.exception = exception;
 }

 /**
  * Enumeration of Result States: OK and EXCEPTION
  */
 public enum ResultState implements Serializable {
  OK(0), EXCEPTION(1);
  private int state;
  private ResultState(int state) {
   this.state = state;
  }
```

```
 public int getState() {
  return state;
 }
}
public T getResult() {
 if (ResultState.OK == this.state) {
  return result;
 } else {
  throw new IllegalStateException("Result is in Exception
state");
 }
}
public Exception getException() {
 if (ResultState.EXCEPTION == this.state) {
  return exception;
 } else {
  throw new IllegalStateException("Result is in OK state");
 }
}
public ResultState getState() {
 return state;
}
}
```

- Ein `Command` kann ausgeführt werden. Die abstrakte Basisklasse wandelt die Ergebnisse in `Result`-Objekte um:

```
public abstract class Command<T> {
 public final Result<T> execute(List<Object> parameter) {
  try {
   T result = doCommand(parameter);
   return new Result<T>(result);
  } catch (Exception e) {
   return new Result<T>(e);
  }
 }
 protected abstract T doCommand(List<Object> parameter) throws
Exception;
}
```

- Eine `CommandMap` hält eine `Map` von `Command`-Implementierungen und stellt diese zur Verfügung:

```
public class CommandMap {

 private HashMap<String, Command> commandMap;
 {
  commandMap = new HashMap<String, Command>();
 }
 @SuppressWarnings("unchecked")
 public <T> Command<T> get(String commandString){
  return commandMap.get(commandString);
 }
 public void addCommand(String commandString, Command command){
  commandMap.put(commandString, command);
 }
 public void removeCommand(String commandString){
  commandMap.remove(commandString);
 }
 public boolean hasCommand(String commandString){
  return commandMap.containsKey(commandString);
 }
 public Set getCommandStrings(){
  return commandMap.keySet();
 }
}
```

4.2.3.3 Beispiel: BooksService

Um den Service-to-Worker zu verwenden muss jeder Service eine Reihe von `Command`-Ereiterungen zur Verfügung stellen. Im Beispiel des `BooksService` sind dies:

Eine allgemeine Basisklasse, die eine Referenz auf den `BooksService` hält:

```
public abstract class BaseBookCommand<T> extends Command <T>{
 private BooksService booksService;
 public void setBooksService(BooksService booksService) {
  this.booksService = booksService;
 }
 protected BooksService getBooksService() {
  return booksService;
 }
}
```

Pro Service-Methode gibt es eine eigene Command-Implementierung.

Buch löschen:

```
public class DeleteBookByIsbnCommand extends BaseBookCommand {
 public Object doCommand(List parameter) throws Exception {
  getBooksService().deleteBookByIsbn((String) parameter.get(0));
  return Result.VOID;
 }
}
```

Alle Bücher suchen:

```
public class FindAllBooksCommand extends
BaseBookCommand<Collection<BookListValue>> {
 public Collection<BookListValue> doCommand(List parameter)
throws BookException {
  return getBooksService().findAllBooks();
 }
}
```

Ein bestimmtes Buch suchen:

```
public class FindBookByIsbnCommand extends
BaseBookCommand<BookValue> {
 public BookValue doCommand(List<Object> parameter) throws
BookException {
 return getBooksService().findBookByIsbn((String)
parameter.get(0));
 }
}
```

Ein Buch neu anlegen:

```
public class NewBookCommand extends BaseBookCommand<String> {
 public String doCommand(List parameter) throws BookException {
 return getBooksService().newBook((String) parameter.get(0));
 }
}
```

Ein Buch aktualisieren:

```
public class UpdateBookCommand extends
BaseBookCommand<BookValue> {
 public BookValue doCommand(List parameter) throws BookException
{
 return getBooksService().updateBook((BookValue)
parameter.get(0));
 }
}
```

Der `BooksCommandController` hält alle `Command`-Objekte innerhalb einer `CommandMap`:

```
public class BookCommandController{
 private CommandMap commandMap;
 private BooksService booksService;
 public void setBooksService(BooksService booksService) {
 this.booksService = booksService;
 }
 public void init() {
 commandMap = new CommandMap();
 NewBookCommand newBookCommand = new NewBookCommand();
 newBookCommand.setBooksService(booksService);
 DeleteBookByIsbnCommand deleteBookByIsbnCommand = new
DeleteBookByIsbnCommand();
```

```
 deleteBookByIsbnCommand.setBooksService(booksService);


 FindAllBooksCommand findAllBooksCommand = new
FindAllBooksCommand();
 findAllBooksCommand.setBooksService(booksService);
 FindBookByIsbnCommand findBookByIsbnCommand = new
FindBookByIsbnCommand();
 findBookByIsbnCommand.setBooksService(booksService);
 UpdateBookCommand updateBookCommand = new UpdateBookCommand();
 updateBookCommand.setBooksService(booksService);
 commandMap.addCommand("newBook", newBookCommand);
 commandMap.addCommand("deleteBookByIsbn",
deleteBookByIsbnCommand);
 commandMap.addCommand("findAllBooks", findAllBooksCommand);
 commandMap.addCommand("findBookByIsbn",
findBookByIsbnCommand);
 commandMap.addCommand("updateBook", updateBookCommand);
 }
 public <T> Result<T> execute(String commandString, List<Object>
params){
 Command<T>  command = commandMap.get(commandString);
 Result<T> result = command.execute(params);
 return result;
 }
}
```

Die Aufgabe der Message Facade ist nun fast trivial: Auslesen der Kommando-Zeichenkette und der Parameter-Liste aus der Message und Rücksenden des erzeugten Resultats:

```
public class BooksServiceMessageFacadeBean extends
 BaseRespondingMessageDrivenBean {
 private BookCommandController bookCommandController;
 private BooksService booksService;
 public BooksServiceMessageFacadeBean() {
 super("ConnectionFactory");
 }
 private static final long serialVersionUID = 1L;


 public ObjectMessage getResponse(Session session, ObjectMessage
message)
  throws JMSException {
```

```
 String command = message.getStringProperty("command");
 List<Object> parameter = (List<Object>) message.getObject();
 Result result = bookCommandController.execute(command,
parameter);
 ObjectMessage response = session.createObjectMessage();
 response.setObject(result);
 return response;
 }
 public void initMessageDrivenBean() {
 booksService = new MapBooksService();
 bookCommandController = new BookCommandController();
 bookCommandController.setBooksService(booksService);
 bookCommandController.init();
 }
}
```

Der Business Delegate gibt je nach Status des empfangenen Resultats entweder das eigentliche Ergebnis zurück oder wirft die entsprechende Exception:

```
public class BooksServiceJmsBusinessDelegate extends
 BaseJmsBusinessDelegate implements BooksService {
 public BooksServiceJmsBusinessDelegate() {
 super("ConnectionFactory", "queue/C");
 }
 public String newBook(String title) throws BookException {
 ArrayList<Object> parameter = new ArrayList<Object>(1);
 parameter.add(title);
 return (String) executeCommand("newBook", parameter);
 }
 public BookValue findBookByIsbn(String isbn) throws
BookException {
 ArrayList<Object> parameter = new ArrayList<Object>(1);
 parameter.add(isbn);
 return (BookValue) executeCommand("findBookByIsbn",
parameter);
 }
 public Collection<BookListValue> findAllBooks() throws
BookException {
 return executeCommand("findAllBooks", Collections.EMPTY_LIST);
 }
```

```
public BookValue updateBook(BookValue bookDetailValue) throws
BookException {
 ArrayList<Object> parameter = new ArrayList<Object>(1);
 parameter.add(bookDetailValue);
 return (BookValue) executeCommand("updateBook", parameter);
}
public void deleteBookByIsbn(String isbn) throws BookException
{
 ArrayList<Object> parameter = new ArrayList<Object>(1);
 parameter.add(isbn);
 executeCommand("deleteBookByIsbn", parameter);
}
private <T> T executeCommand(String commandString, List
parameter)
 throws BookException {
 Result<T> result = sendCommand(commandString, parameter);
 if (result.getState().equals(Result.ResultState.OK)) {
 return result.getResult();
 } else {
 Exception e = result.getException();
 if (e instanceof RuntimeException) {
 throw (RuntimeException) e;
 } else {
 throw (BookException) e;
 }
 }
}
}
```

4.2.4 Message Facade und Klassendiagramm

Die Message Facade übernimmt im Klassendiagramm eine der Remote Session Facade entsprechenden Rolle. Sie wird in der Regel an die Local Session Facade delegieren, um die darin konfigurierten Dienste nicht redundant konfigurieren zu müssen. Prinzipiell ist es aber auch möglich, dass die Message Facade selber die Transaktionssteuerung übernimmt, da auch eine MessageDrivenBean den Zugriff auf den Transaktionsmanager des Applikationsservers hat. Dies entspricht einer Architektur, in der die Verteilung ausschließlich über JMS realisiert werden soll.

4.2.5 Dokumenten-orientierte Services

Das Command-Pattern erlaubt es auf sehr einfache **Weise**, Services mit neuer Funktionalität zu erweitern:

- Die Einführung einer neuen Methode erfordert keine Änderung der Signatur der Message Facade. Dies wäre ja auch unmöglich, da ausschließlich eine `onMessage`-Methode vorhanden ist. Es muss nur in der `CommandMap` ein neues Kommando registriert werden. Dies ist aber bei geeigneter Programmierung sogar zur Laufzeit möglich.
- Ein Kommando kann von einer ganzen Kette verarbeitender Instanzen behandelt werden, von denen jede einen bestimmten Teil der Fachlogik modular getrennt ausführt. Diese Idee ist das im Zusammenhang mit Command eingeführte Entwurfsmuster „Chain of Command“. Ebenso wie das Hinzufügen neuer `Command`-Instanzen können somit auch ganze Verarbeitungsschritte dynamisch zur Laufzeit hinzugefügt oder entfernt werden.
- Ein `Command`-Objekt kann sehr einfach an komplett andere Services weitergeleitet werden. Es ist an dieser Stelle nur ein Umkopieren/Ergänzen der Parameter-Liste notwendig[12].

Alle diese Argumente sind sogar unabhängig von JMS und lassen sich zu dem Begriff „Dokumenten-orientierte Schnittstelle“ verallgemeinern. Kennzeichen einer Dokumenten-orientierten Schnittstelle ist, dass statt typisierter Methoden-Signaturen nur eine einzige Signatur angeboten wird:

```
public ResultDocument execute(ParameterDocument)
```

[12] Wäre in der Basisklasse statt einer Liste eine Map verwendet worden könnte diese ohne Kopie durch weitere Parameter ergänzt werden, was für dieses Szenarium sicherlich vorteilhaft ist.

`ResultDocument` und `ParameterDocument` haben keinen festen Typ und damit keine feste Struktur sondern enthalten flexibel Informationen. Eine Server-seitige Verarbeitung nimmt sich aus dem `ParameterDocument` die Informationen, die für seine Verarbeitung relevant sind heraus und ignoriert den Rest. Für eine Weiterverarbeitung in einer „Chain of Command“ kann die Verarbeitung jederzeit weitere Informationen ergänzen, die dann in einer Folgeverarbeitung benötigt werden.

Wird als Dokument-Typ XML verwendet, so kann durch die Vorgabe eines XML-Schemas zumindest zur Laufzeit eine Validierung erfolgen. Dem Java-Typ, das der Compiler prüft, entspricht hier dann das XML-Schema, das der Parser validiert.

4.2.6 Dokumenten-orientierte Remote Facade

Selbstverständlich kann eine Dokumenten-orientierte Fassade auch mit einer Session Bean realisiert werden. Dazu wird einfach die obige Signatur

```
public ResultDocument execute(ParameterDocument)
```

übernommen. Dies kann bereits im Platform Independent Model geschehen oder aber in der Fassaden-Signatur des Platform Specific Models.

Im ersteren Fall ist die Service-Schnittstelle selbst bereits Dokumenten-orientiert modelliert. Im zweiten Fall sind die Dokumente als Transfer Documents aufzufassen. Der zweite Fall kann eine zentrale Session Facade alle ankommenden Anfragen zentral verwalten. Dies entspricht dem Design Pattern EJB Command[13]. Die sich daraus ergebenden Vorteile liegen auf der Hand:

- Konsistentes Programmiermodell für Message Facade und Remote Facade
- Zentraler Einstiegspunkt für das dynamische Einhängen weiterer Dienste
- Dynamische Erweiterungen/Änderungen von Services ohne erneutes Erzeugen der Fassaden. Es genügt wie bei der Message Facade die Verwaltung von `Commands` in der `CommandMap`.

[13] Siehe http://www.theserverside.com

4.3 Die Http Facade

Eine Verteilung der Anwendung ist auch über das http-Protokoll möglich[14]. Hier dient der `javax.servlet.http.HttpServletRequest` als Transfer Document auf und wird von einem Servlet entgegen genommen.

Durch die Einführung des Service-to-Worker ist eine Realisierung sehr einfach: Die http Facade analysiert den Request und delegiert genau so wie die Message Facade an diesen weiter:

```
public class BooksServiceHttpFacadeServlet extends HttpServlet
{
 private static final long serialVersionUID = 1L;
 private BookCommandController bookCommandController;
 private BooksService booksService;
 protected void doGet(HttpServletRequest request,
  HttpServletResponse response) throws ServletException,
IOException {
 doRequest(request, response);
 }
 protected void doPost(HttpServletRequest request,
  HttpServletResponse response) throws ServletException,
IOException {
 doRequest(request, response);
 }
 private void doRequest(HttpServletRequest request,
  HttpServletResponse response) throws ServletException,
IOException {
 String commandString = request.getParameter("command");
 ArrayList<Object> params = new ArrayList<Object>();
 String paramString;
 int counter = 0;
 while ((paramString = request.getParameter("param" + counter))
!= null) {
 params.add(paramString);
 counter++;
 }
```

[14] Dies bedeutet natürlich nicht, dass es sich hierbei um eine Web Anwendung handelt. Es werden ausschließlich die Service-Aufrufe über http getunnelt, der Client bleibt beispielsweise eine Swing-Anwendung.

```
 Result result = bookCommandController.execute(commandString,
params);
 ObjectOutputStream oos = new ObjectOutputStream(response
  .getOutputStream());
 oos.writeObject(result);
 }
 public void init() throws ServletException {
 booksService = new MapBooksService();
 bookCommandController = new BookCommandController();
 bookCommandController.setBooksService(booksService);
 bookCommandController.init();
 }
}
```

Der Business Delegate baut einen geeigneten URL-Request auf und liest aus dem `InputStream` das Resultat:

```
public class BooksServiceHttpBusinessDelegate implements
BooksService {
 public String newBook(String title) throws BookException {
  return (String) executeCommand("newBook", "&param0=" + title);
 }
 public BookValue findBookByIsbn(String isbn) throws
BookException {
  return (BookValue) executeCommand("findBookByIsbn", "&param0="
+ isbn);
 }
 public Collection<BookListValue> findAllBooks() throws
BookException {
  return executeCommand("findAllBooks", "");
 }
 public BookValue updateBook(BookValue bookDetailValue) throws
BookException {
  throw new UnsupportedOperationException();
  }
 public void deleteBookByIsbn(String isbn) throws BookException
{
  executeCommand("deleteBookByIsbn", "&param0=" + isbn);
 }
 @SuppressWarnings("unchecked")
 private <T> T executeCommand(String command, String params)
  throws BookException {
```

```
 String urlString =
"http://localhost:8080/books/BooksHttpFacade?command="
  + command + params;
 Result<T> result = null;
 try {
 URL url = new URL(urlString);
 URLConnection connection = url.openConnection();
 ObjectInputStream ois = new ObjectInputStream(connection
  .getInputStream());
 result = (Result<T>) ois.readObject();
 ois.close();
 } catch (Exception e) {
 e.printStackTrace();
 throw new RuntimeException(e.getMessage());
 }
 if (result.getState().equals(Result.ResultState.OK)) {
 return result.getResult();
 } else {
 Exception e = result.getException();
 if (e instanceof RuntimeException) {
  throw (RuntimeException) e;
 } else {
  throw (BookException) e;
 }
 }
}
```

Hier ist zu beachten, dass die Serialisierung des `BookValue` nicht ausimplementiert wurde. Sollen komplexe Objektbäume übertragen werden ist eine eigene Implementierung nicht wirklich sinnvoll, da auch im Open Source-Bereich dafür genügend Algorithmen existieren. Genannt sei hier das Burlap/Hessian-Verfahren.

4.4 Die REST Facade

Der Aufwand der Implementierung einer http-Facade kann mit Jax-RS, der Java-Spezifikation zur Realisierung von RESTful WebServices komplett vereinfacht werden. Das Mapping des URL-Requests auf Java-Methoden sowie die Encodierung des Ergebnis-Objekts in ein Dokumenten-Format wird über Annotations definiert und vom RESTful-Framework[15] realisiert.

```
import java.io.StringWriter;

import javax.ejb.EJB;
import javax.ejb.Stateless;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Marshaller;

import org.javacream.publishing.business.PublishingController;
import org.javacream.publishing.entities.Book;

//@Named("PublishingRestBean")
//@ManagedBean
@Stateless
@Path("books")
@Produces(MediaType.TEXT_XML)
public class PublishingRestBean {

 @EJB
 private PublishingController publishingController;

 @GET
```

[15] JAxRS ist nicht Bestandteil der Jee 6-Spezifikation, kann jedoch problemlos hinzugefügt werden. Neben der Referenz-Implementierung Jersey, existiert eine ganze Reihe von Open-Source-Bibliotheken.

```
@Path("/{isbn}")
public String findBookByIsbn(@PathParam("isbn") String isbn) {
 Book book = publishingController.findBookByIsbn(isbn);
 try {
  JAXBContext context = JAXBContext.newInstance(Book.class);
  Marshaller marshaller = context.createMarshaller();
  StringWriter stringWriter = new StringWriter();
  marshaller.marshal(book, stringWriter);
  return stringWriter.toString();
 } catch (JAXBException e) {
  e.printStackTrace();
  return e.getMessage();
 }

}

}
```

4.5 Die SOAP Facade

Service-Schnittstellen sind auf Grund ihrer Definition in der Regel bereits kompatibel zur Web Service Spezifikation. Das Ansprechen eines Services über das SOAP-Protokoll erfordert deshalb nur sehr wenig Aufwand. Die im Rahmen der JaxWS-Spezifikation enthaltenen Annotations ermöglichen es, beispielsweise eine vorhandene Session Bean sofort zu einem Web Service zu machen:

```
@WebService(name = "EndpointInterface", targetNamespace =
"http://org.javacream.ws/keygenerator", serviceName =
"KeyGeneratorWebService")
@SOAPBinding(style = SOAPBinding.Style.RPC)
@Stateless
public class KeyGeneratorSoapFacadeBean implements
RemoteKeyGenerator {
 private RandomKeyGenerator randomKeyGenerator;
 @PostConstruct
 public void initFacade() {
  randomKeyGenerator = new RandomKeyGenerator();
 }
    @WebMethod
 public String next() {
  return randomKeyGenerator.next();
 }
 @Override
 public String toString() {
  return "KeyGeneratorEjb3SessionBean@" + hashCode();
 }
}
```

Der Zugriff auf die Facade erfolgt dann wie Üblich über einen Business Delegate. Dieser holt sich in diesem Fall die Referenz auf den Web Service über RPC:

```
       try {
            QName qname = new
QName("http://org.javacream.ws/keygenerator ",
                    "KeyGeneratorWebService");
            String urlstr =
"http://localhost:8080/javacream/KeyGeneratorSoapFacadeBean?wsd
l";
            URL url = new URL(urlstr);


            ServiceFactory factory =
ServiceFactory.newInstance();
            Service service = factory.createService(url,
qname);
            RemoteKeyGenerator generator = (RemoteKeyGenerator)
service.getPort(RemoteKeyGenerator.class);
        } catch (Exception e) {
            e.printStackTrace();
            fail(e.getMessage());
        }
    }
}
```

Selbstverständlich kann die SOAPFacade auch die Aufgaben einer echten Remote Facade übernehmen, also beispielsweise spezielle Transfer Objekte erzeugen (hier wahrscheinlich XML-Dokumente), Signaturen ändern etc. Insbesondere in Interoperablen Architekturen, in denen beispielsweise ein Perl- oder .NET-Client den Web Service aufrufen will ist es häufig notwendig, aus komplexen Objektbäumen einfachere Dokumente zu erzeugen.