1 Definition von JavaBeans

Mit JavaBeans definiert Sun ein Komponentenmodell, das Softwareherstellern ermöglichen soll, Javakomponenten zu erstellen, die von den Endusern möglichst einfach angepaßt und zu Applikationen zusammengefügt werden können. Als Erweiterung der Programmiersprache Java spiegeln sich viele derer Merkmale, wie z.B. Portabilität und Sicherheit, in den Beans wider.

Ein JavaBean ist eine wiederverwendbare Softwarekomponente, die in einer Entwicklungsumgebung visuell manipuliert werden kann.

Diese Definition erinnert stark an bekannte Entwicklungssysteme, welche die Idee der komponentenbasierten Softwareerstellung verfolgen (vgl. z.B. Borland Delphi oder Microsoft Visual Basic). Das von Sun vorgelegte Konzept der JavaBeans verfeinert jedoch vorhandene Ansätze (z.B. umfaßt der Begriff der Persistenz hier nicht nur die Möglichkeit der bleibenden Veränderung aus der Entwicklungsumgebung heraus, sondern erlaubt einem Bean auch zur Laufzeit Zustandsänderungen permanent zu machen) und erweitert diese (z.B. ermöglicht der Mechanismus der Introspection nicht nur den Sicherheitscheck vor der Ausführung sondern auch die einfachere Integration in Entwicklungsumgebungen).

Es existieren keinerlei Beschränkungen den Leistungsumfang betreffend: Möglicherweise findet ein Bean seine Aufgabe darin, Loginzeiten mitzuschreiben, ein anderes stellt ein Spreadsheet dar und ein drittes ist ein simpler Button auf einer HTML Seite.

2 Charakterisierung

2.1 Funktionale Einheit

Ein Bean als funktionale Einheit ist für einen bestimmten Verwendungszweck konzipiert. Die Eigenschaften und das Erscheinungsbild werden mit der Funktionalität abgestimmt.

2.2 Visuelle Bearbeitung

Beans können in einer Entwicklungsumgebung visuell manipuliert, d.h. die Größe und Position in einem Fenster kann ohne Programmieraufwand verändert werden (ein Bean muß nicht notwendigerweise eine GUI-Repräsentation zur Laufzeit besitzen), Properties können verändert werden und mehrere Beans können miteinander ereignisbezogen verknüpft werden.

2.3 Unterschiedliche Erscheinung/Verhaltensweise

2.3.1 Verschiedene Runtimeumgebungen

Das Verhalten eines Beans kann auch von System zu System verschieden sein: So ist es z.B. möglich eine Komponente zu schreiben, die zwischen Clients und mehreren Servern wandert und dabei unterschiedliche Aufgaben erfüllt. Bei der Ausführung auf den Clients sammelt es Daten, auf den Servern wertet es diese in Verknüpfung mit anderen Daten aus. Hier sei noch angemerkt, daß das Bild, wie sich die Komponente dem Benutzer präsentiert, gravierend variieren kann: Zeigt das Bean in einem Fall ein Formular an, in das Daten eingetragen werden soll, öffnet es bei der Ausführung am Server gar kein Fenster.

2.3.2 Unterschiedliches Verhalten Design-/Runtime

In der Entwicklungsumgebung legen Beans ein anderes Verhalten zu Tage als zur Laufzeit. So kann ein Bean den Programmierer bei seiner Arbeit mit Wizards unterstützen, ohne daß ein ähnliches Verhalten zu einem späteren Zeitpunkt im Echtbetrieb auftritt. Wir unterscheiden das Laufzeitverhalten vom Verhalten in der Entwicklungsumgebung.

2.4 Persistenz

Ihren internen Zustand können Beans abspeichern, sodaß sie zu einem späteren Zeitpunkt in ähnlicher Erscheinungsweise und mit ähnlichem Verhalten wider aktiviert werden können. Die Persistenz wird über Serialization/Externalization-Mechanismen erreicht, die von Beans unterstützt werden müssen.

Beans können auch andere Beans enthalten. In diesem Fall sollten deren Zustände als Teil des eigenen Zustands gespeichert werden.

2.5 Plattformunabhängigkeit

Da Beans auf Java basieren, erben sie die Eigenschaft der Plattformunabhängigkeit. Beim Design der JavaBeans-APIs hat Sun darauf geachtet, die Integration in vorhandene Komponentenmodelle unterschiedlicher Plattformen zu ermöglichen. So existieren Schnittstellen zu OpenDoc, OLE/COM/ActiveX und LiveConnect. Die Integration eines Beans in einen plattformspezifischen Container (z.B. Word) wird damit möglich.

2.6 Sicherheit

Für Beans gelten die gleichen scharfen Sicherheitskriterien wie für Java. Die Möglichkeiten eines Beans hängen davon ab, ob es als Application oder Applet läuft. Untrusted Applets haben eingeschränkten Zugriff auf Dateien, nur eingeschränkten Einblick in die Laufzeitumgebung und dürfen sich nicht in die graphische Oberfläche der Elternapplikation integrieren (kein GUI-Merging).

3 Vorteile von JavaBeans

Aus der modularen Konzeption resultieren für Komponentenentwickler wie auch für Komponentenanwender eine Reihe von Vorteilen.

3.1 Vorteile für Komponentenentwickler

Der Entwickler muß sich an spezifizierte Schnittstellen halten. JavaBeans haben einen bestimmten Aufbau und werden genormt vertrieben. Daher muß für den Anwender nicht noch zusätzlich weitere Unterstützung implementiert werden, da dieser ohne weiteren Aufwand das JavaBean einsetzen kann.

Da ein JavaBean eine logische Einheit bildet, kann vorhandener Code bei der Entwicklung wiederverwendet werden. Zum einen können Methoden teilweise ohne Änderungen von einem anderen JavaBean übernommen werden (z.B. get-/setproperty Methoden, siehe Abschnitt *Properties*), und zum anderen können komplexere JavaBeans wiederum aus mehreren simplen zusammengesetzt werden.

Die Auslieferung von JavaBeans gestaltet sich einfach, da Sun die Pakete, in die Beans verpackt werden, vorgibt. Eine JAR-Datei (siehe Abschnitt Packaging) enthält alles, was zu einem Bean gehört.

3.2 Vorteile für Anwender und Entwickler

Für den Benutzer von JavaBeans stellt sich im wesentlichen die einfache Anwendbarkeit heraus.

In einer grafischen Entwicklungsumgebung können via drag´n´drop Komponenten in Formulare eingesetzt und visuell manipuliert werden. Reaktionen auf seine Umwelt können genauso einfach definiert werden wie sein Erscheinungsbild. Für simplere Beans muß gar nicht erst in den Code eingegriffen werden, es reicht wenn es plaziert und konfiguriert wird.

Die Integration neuer JavaBeans in die Entwicklungsumgebung stellt sich ebenso einfach dar. Der Benutzer gibt der IDE nur das Bean in Form einer JAR-Datei (z.B. aus dem Internet oder von einem kommerziellen Vertreiber) an, diese wird automatisch analysiert und der Inhalt in die Komponentenbibliothek übernommen.

Ein JavaBean hat weiters noch die Möglichkeit, dem Benutzer zusätzliche Unterstützung zu geben. Es kann z.B. mittels eines Wizards eine einfachere Konfigurationsmöglichkeit bieten als direkt durch Properties. Solcher Code ist dann nur in der IDE verfügbar, wird aber nicht in das fertige Javaprogramm übernommen.

4 Merkmale von JavaBeans

JavaBeans haben eine Reihe von Merkmalen und Eigenschaften, die nicht zuletzt für deren einfache Handhabung verantwortlich sind.

Ein Bean kann *beliebig komplex* sein. Meist werden sie für einfache, kompakte Teilaufgaben erstellt und eingesetzt. Typische Beispiele sind Elemente grafischer Benutzeroberflächen wie z.B. Buttons, Eingabefelder oder Scrollbars. Solche einzelnen Elemente lassen sich nun wiederum zu größeren Beans zusammenstellen, wie z.B. ganzen Benutzerdialogen, Datenbankschnittstellen oder Bildbetrachtern. Von Seiten der Spezifikation gibt es keine Maximalgröße von Beans.

So wie jede andere Klasse auch besitzen JavaBeans *Properties* und *Methoden*. Properties sind form- und sinngebende Eigenschaften, die Erscheinungsbild und Stati eines Beans definieren. Methoden sind funktionale Einheiten (Funktionen), die Änderungen und Übergänge in den Properties durchführen. Mehr dazu in den folgenden Abschnitten.

Da in einer objektorientierten Umgebung die einzelnen Instanzen nicht direkt miteinander in Kontakt treten können, sind *Events* implementiert worden, die Nachrichten über Ereignisse zwischen Objekten (JavaBeans, andere Javaobjekte, ..) austauschen.

Um Entwicklungsumgebungen den Einblick in JavaBeans zu ermöglichen, wurde der *Introspection*-Mechanismus eingeführt. Der Designer eines Beans kann diesem, explizit oder implizit, Information über dessen Aufbau mitgeben, welche von der IDE ausgelesen werden, damit ein JavaBean korrekt dargestellt und benutzt werden kann. So werden z.B. die Properties eines Beans bestimmt.

Die Option eines JavaBeans, seine eigenen Editiermöglichkeiten mitzubringen, wird *Customization* genannt. Besitzt ein Bean Datenstrukturen, die schwer oder gar nicht von den Standardroutinen eingelesen oder verändert werden können, so muß es seine eigenen Editoren zur Verfügung stellen (z.B. in Form eines Wizards).

Damit ein JavaBean trotz großem Funktionsunmfangs handhabbar bleibt, hat es zwei wesentliche Erscheinungsbilder: das zur *Designtime*, und das zur *Runtime*. So kann ein Entwickler zur Designtime vermutlich alle Properties einsehen und ändern, während bei der Ausführung sehr wohl auf Kapselungsmöglichkeiten von Objekten Rücksicht genommen werden muß. Auch muß Code, der für Customization benötigt wird, nicht mehr im fertigen Produkt enthalten sein.

5 Unterscheidung Design-/Runtime

5.1 Runtime

Das Laufzeitverhalten ist die applikationsbezogene Funktionalität der Bean. Dieser Teil eines Beans kann mit einer Klasse verglichen werden, die Funktionen, Methoden und Eigenschaften für die Verwendung in einem Programm zur Verfügung stellt.

5.2 Designtime

In der Entwicklungsumgebung muß die Bean die Manipulation des Aussehens und Verhaltens erlauben. Wird der Entwickler mit Wizards unterstützt, kann der Quellcode einer Bean ziemlich umfangreich ausfallen, daher ist eine klare Trennung zwischen Designtime und Runtime erforderlich. So wird sichergestellt, daß eine Bean ausgeliefert werden kann, ohne daß die für die Entwicklung bestimmten (und daher nicht mehr benötigten) Programmteile mitgeliefert werden müssen.

6 Persistence

Da JavaBeans keine statischen Objekte sind, sondern sich durch ihre Properties in einem gewissen Zustand befinden, kann es interessant sein, diesen Zustand abzuspeichern, um später bei einem Neustart wieder in diesen zurückzufinden. Die JavaBeans-Spezifikation sieht dafür zwei Wege vor: *Serialization* und *Externalization*.

6.1 Serialization

Besitzt ein JavaBean keine besonderen Merkmale oder Stati, und können alle relevanten Daten aus den von außen zugänglichen Properties ausgelesen werden, so kann der von Java zur Verfügung gestellte Serialization-Mechanismus eingesetzt werden. Dabei übernimmt Java das Speichern der Properties. Es wird eine `.ser` Datei angelegt, in der die Inhalte der Properties in einem vorgegebenen Format abgelegt werden.

Die wesentliche Einschränkung dieser Methode ist jedoch, daß Java nicht alle relevanten Properties bekannt sein müssen, da durch Kapselung bestimmte Eigenschaften versteckt und von außen nicht zugänglich sein können. Werden diese nicht mitgesichert und wieder zurückgeladen, so funktioniert das Bean eventuell nicht mehr korrekt.

6.2 Externalization

Sollte ein JavaBean mit dem Serialization-Mechanismus nicht mehr auskommen können, so muß es, wenn es sich seinen Status peristent machen will, eigene Funktionen zur Speicherung der Properties und Variablen mitbringen. Da diese selbst codiert werden, haben sie natürlich volle Kontrolle über das Format, und können jene Daten speichern, die auch relevant sind.

Es wird sinnvoll sein, alle standard-Properties mittels Serialization von Java speichern zu lassen, und nur für darüber hinausgehende Eigenschaften zusätzlichen Externalizationcode zu schreiben.

7 Properties

7.1 Überblick über Properties

Properties sind diskrete, benannte Attribute von Beans, welche deren Erscheinungsbild oder Verhalten beeinflussen.

Properties eines Beans können durch Aufruf der zugehörigen Methoden gelesen und geschrieben werden. Die Get-Methode stellt den Wert einer lesbaren Property zur Verfügung, die Set-Methode erlaubt die Änderung des Wertes überschreibbarer Properties. Aus einer Skriptumgebung, in die das Bean eingebunden wird, kann auf die Properties direkt zugeriffen werden. Zuweisungen wie zB. firestone=`fred` werden in Methodenaufrufe umgewandelt. Da das Auslesen einer Property ausschließlich über einen derartigen Aufruf erfolgt, kann ein Property problemlos zum Zeitpunkt des Abrufs berechnete Werte liefern. Auch die Methode, die für das Schreiben einer Property zuständig ist, kann für wertabhängige Folgeaktionen genutzt werden.

Typische Bean-Properties sind persistent, dh. ihr Inhalt wird gespeichert und steht auch beim nächsten Aufruf des Beans zur Verfügung.

In der Entwicklungsumgebung können die Properties innerhalb eines Property-Sheets vom Applikationsentwickler bearbeitet werden. Ein Property-Sheet ist ein Fenster, das die Namen der einzelnen Properties mit ihren zugehörigen Werten anzeigt. Die Werte können in dem Fenster dauerhaft verändert werden.

7.2 Typen von Properties

Neben den Standardproperties, die genau einen Wert aufnehmen können und auf diesen nicht speziell reagieren gibt es Indexed Properties, Bound Properties und Constrained Properties.

7.2.1 Indexed Properties

Indexed Properties können mehrere Werte aufnehmen. Die einzelnen Zellen werden über einen Index vom Typ Java int angesprochen. Die Größe des Arrays wird von der Set-Methode bestimmt. Es besteht auch die Möglichkeit das gesamte Array als solches zu referenzieren.

7.2.2 Bound Properties

Eine Komponente kann ihren Properties die Möglichkeit bieten Zustandsänderungen an andere Beans weiterzuleiten. Properties, die auf die Änderungen hinweisen, werden Bound Properties genannt.

Nachdem eine Bound Property verändert worden ist, meldet es die Änderung dem PropertyChangeListener, der die Änderung weiterleitet.

7.2.3 Constrained Properties

Um Beans die Möglichkeit zu geben, gegen Zustandsänderungen von Properties Einspruch einzulegen, wurden Constrained Properties eingeführt. Constrained Properties kündigen ihre Zustandsänderung an. Beans die damit nicht einverstanden sind legen ihr Veto ein und verhindern damit die Änderung.

Bevor sich eine Constrained Property ändert, meldet sie es dem VetoableChangeListener, der die Ankündigung weiterleitet.

Es ist sinnvoll Constrained Properties auch als Bound Properties zu codieren. Falls andere Beans Einspruch gegen eine bevorstehende Änderung einlegen dürfen, sollten sie auch über erfolgte Änderungen informiert werden.

8 Eventhandling

Damit Instanzen mit einander kommunizieren können, wurden Events eingeführt. Da JavaBeans auch Klassen sind, stehen ihnen sämtliche Eventhandlingmöglichkeiten von Java offen.

Ein Event (*Ereignis*) ist etwas, das passieren kann. Jedes Ereignis wird durch ein Event State Object beschrieben. Als Unterklasse von `java.util.EventObject` definiert, enthält es alle Daten (Properties und Methoden), die in Zusammenhang mit dem Ereignis relevant sind. So können für das Ereignis MouseMovedEvent die neuen Koordinaten des Mauscursors wichtig sein.

Will ein anderes Bean über die Auslösung eines bestimmtes Events benachrichtigt werden, so muß es sich mit einer entsprechenden EventListener Methode registrieren lassen.

Tritt ein Ereignis in Kraft, von dem ein Bean meint, daß auch andere Beans und/oder Objekte in Kenntnis gesetzt werden müssen, so feuert es ein Event (engl.: fire Event). Alle anderen Beans oder Objekte, die mit einem EventListener für das entsprechende Ereignis registriert sind, werden nun benachrichtigt. Sie können auf das Ereignis reagieren, relevante Werte aus dem Event State Object abfragen und sich entsprechend verhalten.

9 Introspection

Soll ein neues JavaBean in die Entwicklungsumgebung integriert werden, stellt sich für Java folgendes Problem: Wie soll es herausfinden, woraus ein JavaBean besteht, welche Properties es besitzt, welche Methoden usw.

Schließlich soll, wenn der Benutzer ein Bean in sein Programm setzen will, eine dem Bean entsprechende Editiermöglichkeit geboten werden. Die Oberfläche muß einen Propertysheet erstellen, der, ähnlich Borland Delphi oder Mircosoft Visual C++, alle Eigenschaften einer Komponente übersichtlich in Form einer Li-

ste anzeigt, und Werteingaben akzeptiert. Mehr dazu wird im Detail im Abschnitt *Customization* beschrieben.

Java muß nun in das Bean hineinschauen, und möglichst viel an Information über das Bean gewinnen. Dieser Vorgang wird Introspection genannt.

Prinzipiell kennt Java zwei Möglichkeiten des Introspection: die *BeanInfo* Klasse und *Reflection*.

9.1 BeanInfo Klasse

Mit der BeanInfo Klasse hat der Entwickler die Möglichkeit, die Attribute seines JavaBeans nach seinen Wünschen bekanntzugeben. Properties, Methoden und Events werden implizit angegeben.

Diese Klasse heißt `java.beans.BeanInfo`, und liefert Daten wie den BeanDescriptor, das Default-Event, das Icon, welches das Bean in der Entwicklungsumgebung symbolisieren soll, die Methoden-Deskriptoren und die Property-Deskriptoren. Um nun Detailinformationen zu erhalten, werden wiederum die Deskriptoren aufgerufen, usw.

Was der Entwickler nicht angibt, ist somit auch Java nicht bekannt und kann auch nicht eingesehen oder editiert werden. Das gibt dem Programmierer zusätzlich zur Kapselung die Möglichkeit, dem Anwender interne Funktionen vorzuenthalten.

9.2 Reflection

Sollte der Entwickler keine BeanInfo Klasse zur Verfügung stellen (Java findet keine), so hat Java noch die Möglichkeit Properties, Methoden und Events an den Design Patterns zu erkennen.

9.2.1 Design Patterns

Design Patterns sind Konventionen, wie Properties, Methoden und Events benannt werden sollten. Es steht dem Entwickler frei, ob er sich an diese Vorgaben hält oder nicht. Wenn sie eingehalten werden, kann Java selbsttätig (fast) alle relevanten Attribute aus dem Bean-Code herausfinden, und es muß nicht zusätzlich eine BeanInfo Klasse implementiert werden. Es folgen ein paar Beispiele, wie Properties bzw. deren Methoden und Events identifiziert werden können.

Einfache Properties

```
public <PropertyType> get<PropertyName>();
public void set<PropertyName>(<PropertyType> a);
```

Boolean Properties

```
public boolean is<PropertyName>();
```

Indexed Properties

```
public <PropertyElement> get<PropertyName>(int a);
public void set<PropertyName>(int a, <PropertyElement> b);
```

Events

```
public void add<EventListenerType>(<EventListenerType> a);
public void remove<EventListenerType>(<EventlistenerType> a);
```

Werden diese Konventionen eingehalten, so kann Java anhand der Set-/Get-/Add-/Remove-Methoden die Bezeichner *<PropertyName>* und *<EventListenerType>* ermitteln.

Bei einer sauberen Programmierung entsteht dem Entwickler daher kein Mehraufwand wegen einer zusätzlich erforderlichen Definition für die Entwicklungsumgebung. In diesem Fall muß auf die BeanInfo Klasse nur mehr dann zurückgegriffen werden, wenn sich in dem Bean Elemente befinden, die nicht standardmäßig erkannt werden können.

10 Security

Wie auch in Java selbst wird Sicherheit bei JavaBeans besonders sorgfältig behandelt. Generell gilt, daß Beans nicht mehr dürfen, als normaler Javacode.

Ein *untrusted Applet*, also ein eingebettetes Javaprogramm, das als nicht vertrauenswürdig eingestuft wird, hat keine Schreib-/Leserechte auf Dateien und darf selbst keine Verbindungen zu anderen Servern aufbauen um Daten zu übertragen. Typische untrusted Applets wären optische Aufbesserungen in HTML-Dokumenten wie animierte Schriften und Buttons, und werden meist ohne explizite Zustimmung des Benutzers gestartet.

Ein *trusted Applet* (eingebettet, aber als vertrauenswürdig klassifiziert) oder eine Java Applikation haben kaum Einschränkungen. Sie dürfen Dateien lesen und schreiben, und Daten an fremde Hosts verschicken. Da diese Anwendungen explizit gestartet oder bestätigt werden müssen, kann eine Verwendung kaum unabsichtlich passieren, und die Möglichkeiten der Programme sind gerechtfertigt.

Im Zusammenhang mit JavaBeans sind folgende Aspekte von besonderer Bedeutung:

- Entwickler sollten zur Designtime beliebigen Zugriff auf Objekte, Beans usw. besitzen. Zur Runtime allerdings sollten die Möglichkeiten ebenso beschränkt sein wie für normale Anwender. Das gilt für das Bean natürlich ebenfalls.
- Persistence setzt gewisse Rechte voraus. Damit sich ein Bean initialisieren kann, muß es entsprechende Informationen aus Dateien und von Servern laden können.
- Beans können auch innerhalb anderer Beans benutzt werden. Dabei ist darauf zu achten, daß durch das Konglomerat mehrerer Beans nicht Zugriffe erlaubt werden, die einzeln nicht gestattet sind.

11 Customization

Unter Customization versteht man die Manipulation des Aussehens und Verhaltens eines JavaBeans. Diese kann auf zwei Arten erfolgen: über einen Property Editor oder eine Customizer Class.

11.1 Property Editor

Der Property Editor ist ein Fenster, in dem alle Properties mit ihren Werten aufgelistet werden. Dieses Fenster wird im allgemeinen von der Entwicklungsumgebung erstellt und angezeigt. Es besteht allerdings auch die Möglichkeit, daß das Bean seinen Property Editor selbst mitbringt. Darin können die Property-Werte vom Benutzer modifiziert werden. Diese Art der Customization eignet sich für kleine bis mittlere Beans. Für die Realisierung ist praktisch kein Aufwand erforderlich, Konfigurationen von komplexeren Beans können allerdings unübersichtlich werden.

Das Auffinden eines Property Editors durch die Entwicklungsumgebung erfolgt in mehreren Schritten: Zunächst wird überprüft, ob im PropertyEditorManager ein Editor explizit registriert wurde. Ist dies nicht der Fall, wird nach einer Klasse <Propertyname>Editor (das ist die empfohlene Art Property Editoren beizupacken) gesucht. Existiert diese nicht, folgt die Suche nach einer Klasse , deren Name sich aus dem Eintrag einer Suchliste (default: java.beans.editors) und dem letzten Teil des Namens der Klasse, in der die Funktionalität codiert ist, zusammensetzt. Die Suchliste kann von der Entwicklungsumgebung durch ihre eigene ersetzt werden.

Der PropertyEditorManager von JavaBeans enthält vordefinierte Editoren für die Java Standardtypen. Diese Editoren können von der Entwicklungsumgebung genutzt werden.

11.2 Customizer

Um auch größere Beans rasch und sinnvoll adaptieren zu können, kann der Programmierer der Bean eine Customizer Class vorsehen. Dabei handelt es sich um eine AWT Komponente deren Aufgabe darin besteht die Customization zu unterstützen (Wizzard). Die Customizer Class kann auch auf den Property Editor zurückgreifen.

Damit die Customizer Class gefunden werden kann ist die Existenz einer BeanInfo Klasse zwingende erforderlich.

Um die Einstellungen persistent zu machen, bedient sich die Entwicklungsumgebung der Serialization/Externalization.

12 Packaging

Ist ein JavaBean fertig entwickelt, muß es noch den Weg vom Entwickler zum Anwender finden. Dazu hat Sun das Packaging spezifiziert.

Die seit dem JDK 1.1 unterstützen JAR-Dateien (Java Archive), die im wesentlichen eine Erweiterung der ZIP-Dateien sind, dienen dabei als Packung.

Alle für ein Bean notwendigen Dateien werden in eine Datei gepackt, und optional mit einem beschreibenden Manifest-File ergänzt.

Ein solches Archiv sollte enthalten:

- den eigentlichen Bean-Code, als `.class` Datei
- Serializationinformation, als `.ser` Datei (optional)
- Dokumentation und Hilfe als `.html` Datei (optional)
- Sämtlichen benötigten Ressourcen wie Bilder, Sounds, Videos, ...

Somit muß nur mehr eine einzige Datei für ein gesamtes Bean, sei es auch noch so komplex, vertrieben werden. Entwicklungsumgebungen können dieses Archiv meist automatisch einlesen, das enthaltene Bean extrahieren und in die eigene Komponentenbibliothek übernehmen. Der Anwender muß sich somit bei korrekter Ausführung nicht mehr mit der Installation herumschlagen.

12.1 Manifestdatei

Zusätzlich zu den oben aufgeführten Dateien kann ein JAR-Archiv noch eine Manifestdatei enthalten. Diese muß den Namen `META-INF/MANIFEST.MF` tragen, und enthält Informationen über den Archivinhalt.

Die Manifestdatei ist eine Textdatei, die in Sektionen unterteilt ist. Jede Sektion beschreibt eine Datei des Archivs mit Namen und Abhängigkeiten. Dazu ein Beispiel:

```
Name: ownres/bean.ser

Java-Bean: True

Depends-On: ownres/bean.class
```

Die Angaben sind bis auf die Dateinamen nicht Case-Sensitiv.

13 Verwendete Literatur und Quellenverzeichnis

JavaBeans API Specification 1.01, Sun Microsystems

Using the BDK 1.0, A Tutorial, JavaSoft

Echte Bohne, Komponenten für Java: Beans, c´t 9/97, S. 300

Schnell gebrüht, JavaBeans entwickeln, PC Magazin 7/97, S. 248

Klappe die 1.1ste, Erweiterungen des JDK 1.1, c´t 6/97, S. 364

http://java.sun.com/beans

http://splash.javasoft.com/beans