# 4 WebObjects-Architektur

*»Die größte Herausforderung der Informatik besteht darin, mit ihr kein Durcheinander anzurichten.«*
*(Edsger W. Djikstra)*

WebObjects dient zur Entwicklung verteilter Anwendungen. Die Kommunikation zwischen den einzelnen Komponenten erfolgt über Internet-Standards. Dieses Kapitel behandelt die konzeptuellen Aspekte und Besonderheiten der WebObjects-Architektur und zeigt, welche technischen Implikationen diese mit sich bringen.

**Applikationen = Server-anwendungen**

WebObjects-Applikationen sind robuste, skalierbare Serveranwendungen, die dynamisch HTML generieren und auf verschiedene Datenbanken zugreifen können. Im Gegensatz zu Scripting-Lösungen, wie Tango oder ColdFusion, sind WebObjects-Applikationen allerdings echte, compilierte und vollständig objektorientierte Anwendungen auf Basis einer Vielzahl existierender Klassen. Aus Sicht des Besucher sind diese WebObjects-Anwendungen dagegen ganz normale Sites.

**Grundprinzip: Request & Response**

Wie alle Web-Applikationen, die auf einem HTML-Interface basieren, können auch WebObjects-Applikationen nur auf Anfragen eines Webbrowsers reagieren und nicht selbst tätig werden. Sie unterliegen somit einem Zyklus, der als Request/Response beziehungsweise Anfrage/Antwort bezeichnet wird. Ohne Request kann eine Applikation im Browser des Besuchers keine veränderte Seite anzeigen. Man könnte auch sagen, dass das generierte HTML ab dem Augenblick, in dem es den Server verlässt, fixiert ist. Es findet erst wieder eine Kommunikation mit dem Server statt, wenn der Browser eine weitere Anfrage absetzt, also etwa beim Klick auf Links oder Formular-Buttons.

**Logik im Server**

Aus diesem Grund, und weil die vollständige Logik der Seiten auf dem Server liegt, können solche Applikationen naturgemäß ohne Serverkontakt nicht auf Benutzereingaben reagieren. Wenn Sie also in einem Webshop mit HTML-Interface die Stückzahl im Bestellschein ändern, muss diese Änderung explizit als Formular an den Webserver übertragen werden, um eine neue Seite mit der aktualisierten Gesamtsumme zu bekommen.

**Client-Logik mit JavaScript**

Diese Einschränkung lässt sich mit JavaScript etwas mildern, jedoch sollten Sie es sparsam verwenden, weil Sie im schlimmsten Fall die Logik zweimal implementieren müssen, einmal für den Client und einmal für den Server. Außerdem ist es für manche Projekte gar nicht wünschenswert, Annahmen über die Konfiguration des Web-Browsers zu machen – etwa ob JavaScript ein- oder ausgeschaltet ist.

## 4.1 3-Schicht-Architektur

WebObjects folgt dem Konzept der dreischichtigen Architektur (3-Tier Architecture), bei welcher Benutzeroberfläche, Anwendungslogik und Dienste (Datenbank) vollständig getrennte logische Einheiten bilden.


**Abbildung 4.1** 3-Schicht-Anwendungen trennen konzeptionelle Benutzeroberfläche, Anwendungslogik und fremde Dienste. WebObjects verbindet mit vorgefertigten Komponenten alle drei Teile.

**2-Schicht-Architekturen**

Im Vergleich dazu sind simplere CGI- oder Middleware-Konstrukte wie etwa Perl-Anwendungen mit Datenbankanbindung zumeist zweischichtige Systeme, bei denen die Anwendungslogik ohne Applikationsserver direkt im Webserver stattfindet.

**Grund: »Softwarehygiene«**

Der Grund für eine Mehrschicht-Architektur ist bessere Skalierbarkeit und Lastverteilung. Es können bei Erreichen der Leistungsgrenze weitere Web- und Applikationsserver in Betrieb genommen werden, die sogar auf unterschiedlichen Systemen laufen können. Nebeneffekt dieser klaren Trennung ist eine bessere »Softwarehygiene« im Gesamtsystem.

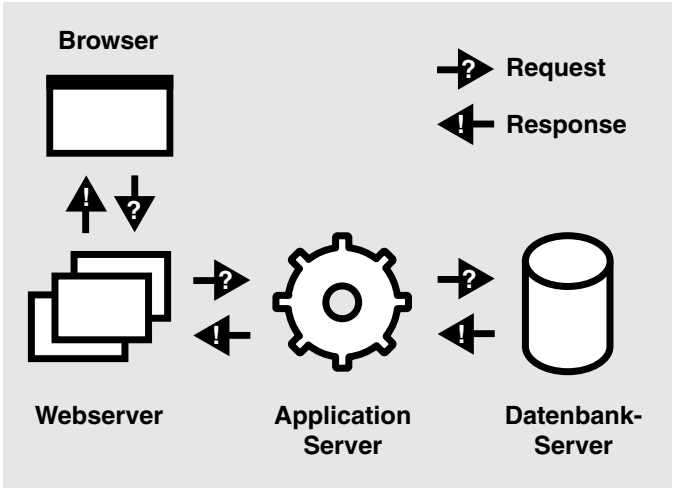


**Abbildung 4.2** Web-, Applikations- und Datenbank-Server bearbeiten der Reihe nach Anfragen eines Clients (Webbrowser) und generieren die Antworten. Die Trennung von Präsentation, Anwendungslogik und Datenquelle bezeichnet man als 3-Schicht-Modell.

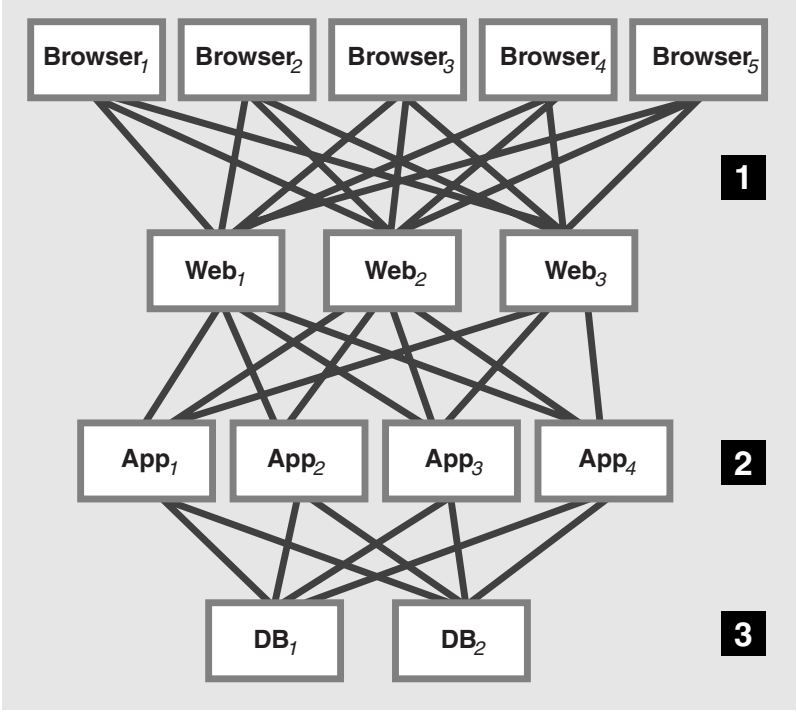


**Abbildung 4.3** Mehrere Web-, Application- und Datenbank-Server erlauben eine Skalierung und damit Verteilung der Last.

## 4.2 Webserver-Adaptor

**Vermittlung**

Da der Webserver aufgrund dieser Trennung eine Anfrage nicht selbstständig beantworten kann, reicht er sie zur Beantwortung an die Applikation weiter. Dafür gibt es im Webserver ein zusätzliches Modul als Vermittlungsschicht zur Applikation, den WebObjects-Adaptor.

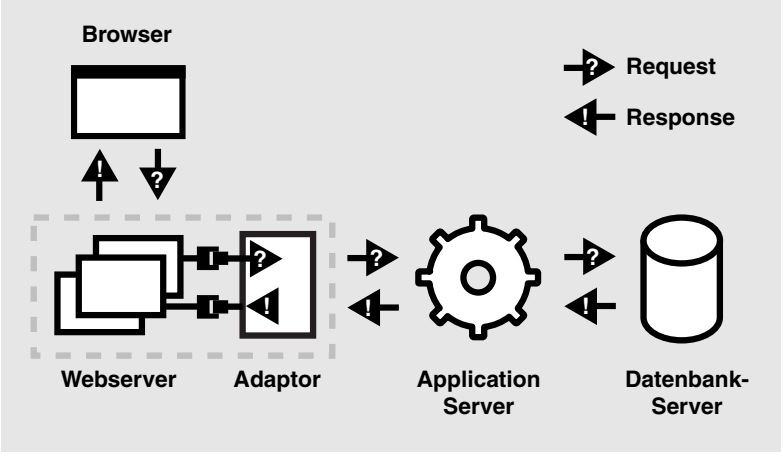


**Abbildung 4.4** Der WebObjects-Adaptor fungiert als Standard-Schnittstelle zwischen Web- und Applikationsserver; er reicht die Anfragen und Antworten weiter.

**Request und Response**

Wie oben bereits gezeigt, können mehrere Applikationsserver in Betrieb sein. Auf jedem können mehrere Applikationen laufen. Deshalb unterhält der WebObjects-Adaptor eine Liste aller laufenden Applikationen und ihrer Portnummern. Er verpackt die HTTP-Anfrage in ein `WORequest`-Objekt und schickt sie über einen normalen HTTP-Request an die zuständige WebObjects-Applikation auf dem entsprechenden Port weiter. In der Applikation findet die Bearbeitung statt, an deren Ende der Adaptor ein `WOResponse`-Objekt zurück erhält, das er wieder in eine HTTP-Standard-Response umwandelt, die an den Browser zurückgeliefert wird.

**Lastverteilung**

Der WebObjects-Adaptor übernimmt bei mehreren Servern oder Instanzen der gleichen Applikation auch die Lastverteilung nach einem einstellbaren Verfahren, beispielsweise »Round Robin«, bei dem jede Instanz der Reihe nach drankommt, »Random«, bei dem zufällig eine Instanz ausgewählt wird oder lastabhängiges Balancing, bei dem der Server mit der geringsten Last die Anfrage bearbeitet. Dies geschieht allerdings nur dann, wenn eine Anfrage nicht fest einer bestimmten Applikationsinstanz zugeordnet werden kann, nämlich genau dann, wenn die Instanznummer in der URL fehlt. Sehen Sie hierzu auch den Abschnitt »URL-Format« im Anhang.

**Skalierbarkeit**

Tatsächlich unterstützt WebObjects die Inbetriebnahme weiterer Applikationsserver und Applikationsinstanzen ohne Beeinträchtigung des laufenden Betriebs. Mit Hilfe einer Steuerungsapplikation, dem WebObjects-Monitor, können weitere Instanzen gestartet werden. Die Information, welche Instanzen auf welchen Servern laufen und auf welchem Port diese erreichbar sind, wird regelmäßig an die Adaptoren der Webserver weitergegeben, die gegebenenfalls ihre Liste aktualisieren. In kurzer Zeit werden so auch neue Applikationen vom WebObjects-Adaptor im Webserver mit Anfragen beschickt.

**Verfügbare Adaptoren**

Es werden native Adaptoren für Netscape API (NSAPI), Information Server API (ISAPI), Web Application Interface (WAI), Apache API mitgeliefert. Native Adaptoren sind effizienter, als der allgemeine CGI-Adaptor, der mit allen Standardservern funktioniert und auch im Quellcode verfügbar ist. So können individuelle Anpassungen für eine konkrete Anwendung vorgenommen werden.

## 4.3 Applikationen

**Application-Konstruktor**

Jede WebObjects-Applikation ist Instanz einer von `WOApplication` abgeleiteten Klasse. Der Konstruktor dieser Klasse wird während der Lebensdauer der Applikation genau einmal gerufen und ist der frühestmögliche Zeitpunkt, zu dem Sie beim Start einer Applikation eingreifen können. Da fast immer die Funktionalität der Basisklasse erweitert und nicht ersetzt wird, sollten Sie bei allen vererbten Methoden die entsprechende Methode der Superklasse einbinden. Im Konstruktor müssen Sie zunächst `super()` rufen, um den Konstruktor der Basisklasse Gelegenheit zur Initialisierung zu geben.

```
public Application () {
   super();
   // hier treffen Sie Ihre Vorbereitungen
}
```

Das Gegenstück zum Konstruktor ist die Methode `terminate()`, die gerufen wird, ehe die Applikation von der Kontrollapplikation heruntergefahren wird. Sie ist der letztmögliche Zeitpunkt, zu dem Sie noch eingreifen können. Leider wird die Methode beim lokalen Testen in ProjectBuilder nicht gerufen, wenn Sie das Stop-Button drücken. In Methoden, die beim Beenden gerufen werden, sollten Sie ebenfalls via `super` die Basisklasse einbinden, wenn Sie deren Funktionalität erweitern, aber nicht ersetzen. Hier ist die Reihenfolge aber umgekehrt, sie müssen erst selbst aufräumen, bevor die Basisklasse aufräumt.

```
public void terminate() {
   // räumen Sie hier Ihre Applikation auf
   super.terminate();
}
```

**Applikationen und Ports**

Die Basisklasse implementiert mit Hilfe der Klasse `WOAdaptor` eine Vermittlungsschicht, welche via HTTP Anfragen entgegennimmt. Somit ist eine Applikation zugleich auch ein minimaler Webserver, der auf einem bestimmten Port lauscht. Das ist in der Regel nicht der Standardport für Webserver (80), sondern eine zufällig vergebene Portnummer. Wenn Sie ein kleines Testprojekt lokal aus ProjectBuilder heraus im so genannten Direct-Connect-Modus starten, ist der Webserver die Applikation selbst. Sie sehen dann in der URL des Browsers die zufällige Portnummer, unter der die Applikation auf HTTP-Anfragen reagiert. Im späteren Betrieb ist die Portnummer natürlich nicht mehr zufällig.

**Adaptoren**

Übrigens wird der Begriff Adaptor in der Apple-Terminologie etwas verwirrend verwendet. Er steht für ein Modul im Webserver, das WebObjects-Anfragen abwickelt und zugleich für die in jeder Applikation vorhandene Instanz der Klasse `WOAdaptor`. Schließlich wird damit auch noch die Abstraktionsschicht zwischen Applikation und relationaler Datenbank bezeichnet. Generell sind Adaptoren die Zwischenschichten, welche die Kommunikation zweier Teile regeln, also zwischen Webserver und Applikation, oder zwischen Applikation und Datenbank.

## 4.4 Sessions

**Session Tracking**

Das im Web verwendete HTTP-Protokoll ist zustandslos, was bedeutet, dass zwischen einzelnen Aktionen des Benutzers keine Verbindung zwischen Browser und Server aufrecht erhalten wird. Deshalb müssen Webanwendungen bei einer Anfrage entscheiden können, ob sie von einem Besucher kommt, der bereits bekannt ist, oder ob es die erste Anfrage ist. Im Fachjargon heißt dieser Mechanismus Session Tracking. Ohne dieses Merkmal müssten Sie sich beispielsweise beim Online-Banking für jede einzelne Seite erneut anmelden, was natürlich nicht akzeptabel sein würde.

**Session-IDs kapseln Benutzer**

Dafür wird ein Benutzer bei seiner ersten Anfrage einer Sitzung (Session) zugeordnet, welche ihn kapselt. Jeder Benutzer hat eine eigene Session und damit der Webserver diese in den Anfragen wieder erkennt, wird in der URL der Seiten eine eindeutige Nummer, die Session-ID, mitgeführt. Alle Anfragen eines einzelnen Anwenders enthalten diese Nummer. Die Session-ID wird von WebObjects automatisch generiert, bei Bedarf kön-

nen Sie einen eigenen Algorithmus verwenden, der höheren kryptographischen Anforderungen gerecht wird. Falls Sie keine Session-ID in der URL haben möchten, können Sie die Session-ID auch über Cookies transportieren.

**WOSession**

Aus Entwicklersicht ist eine Session die Instanz einer Klasse, die von `WOSession` abstammt. Darin wird die Zustandsinformation eines einzelnen Anwenders zwischen den einzelnen Zugriffen gespeichert. Beispielsweise der Inhalt seines Bestellscheins und ob er sich bereits angemeldet hat.

**Session-Konstruktor**

Im Konstruktor sollten Sie wie bei allen vererbten Klassen zuerst den Konstruktor der Basisklasse rufen.

```
public Session() {
   super();
   // hier treffen Sie Ihre Vorbereitungen
}
```

**WOSessionStore**

Zwischen zwei Transaktionen eines Benutzers wird die Session inaktiviert und im Session-Speicher der Applikation, einer Instanz der Klasse `WOSessionStore`, abgelegt. Nach einer einstellbaren Zeit der Untätigkeit (Timeout), also fehlender Kommunikation zwischen Client und Server, wird die Session eines Benutzers gelöscht. Der voreingestellte Zeitraum ist eine Stunde.

**Session-Destruktor**

Das Gegenstück zum Konstruktor ist die Methode `terminate()`, die gerufen wird, ehe das Session-Objekt gelöscht wird:

```
public void terminate() {
   // räumen Sie hier Ihre Session auf
   super.terminate();
}
```

**Explizite Terminierung**

Sie können `terminate()` auch explizit rufen, wenn eine Session beendet werden soll, beispielsweise, wenn ein Anwender seine Bestellung abgeschickt hat oder er auf den Abmelden-Button geklickt hat. Bei einer darauf folgenden Anfrage für eine bereits beendete Session sieht der entsprechende Anwender eine Fehlerseite.

**Warum Timeouts?**

Ohne das Löschen nach einer gewissen Zeit würde die Anzahl der Sessions in einer Applikation immer weiter bis ins Unendliche anwachsen. Außerdem wären sie ein Sicherheitsloch für Online-Banking- und Broking-Applikationen, die aus diesem Grund in der Regel bereits nach wenigen Minuten Untätigkeit die Sitzung beenden.

**Neue Sessions**

Wann wird eine neue Session angelegt? Der erste Zugriff eines Nutzers erzeugt in der Regel eine neue Session. Für eine Applikation namens »MyShop« auf einer Site »www.xyz.de« würde eine URL für den ersten Zugriff so aussehen:

```
http://www.xyz.de/cgi-bin/WebObjects/MyShop.woa
```

Das Adaptor-Modul `WebObjects` reicht die Anfrage an die WebObjects-Applikation `MyShop.woa` weiter. Dieses liegt im im `cgi-bin`-Verzeichnis, sofern der Adaptor als CGI verwendet wird. Da ein Adaptor mehrere verschiedene Applikationen bedienen kann, ist der Name der Applikation in der URL enthalten. Die Extension `.woa` steht für »WebObjects Application«.

**Default Requests**

Weil dieser Request außer dem Applikationsnamen keine zusätzliche Information, etwa Name einer Seite, enthält wird er vom Default Request Handler der Applikation beantwortet. Voreingestellt ist als Default Request Handler der Komponenten-Handler, welcher standardmäßig eine neue Session anlegt und in diesem Session-Kontext die Komponente `Main` zurückliefert, die in jeder Applikation standardmäßig vorhanden ist. Da sich der Anwender nun in einer Session befindet, verändert sich der URL nachfolgender Komponenten. Wenn Sie auf der allerersten Seite einen Verweis zu einer anderen dynamischen Seite haben, wird deren URL bereits die Session-ID enthalten. Er könnte beispielsweise so aussehen:

```
http://www.xyz.de/cgi-bin/WebObjects/MyShop.woa/wo/
7O4000xf500oq800H/4.1
```

**URL-codierte Session-IDs**

Was die einzelnen Teile dieser URLs bedeuten, finden Sie unter »URL-Format«. Entscheidend ist jedoch, dass nun eine Session-ID, nämlich `7O4000xf500oq800H`, enthalten ist. Durch die Speicherung des Anwenders in Form einer Session-ID in der URL benötigt WebObjects im Gegensatz zu klassischen Scripting-Ansätzen, wie JSP und PHP, keine Cookies. Beachten Sie, dass der Name der Komponente `Main` nicht in der URL auftauchen muss, weil sich WebObjects intern für jede Transaktion anhand der Context-ID, in unserem obigen Beispiel 4.1, die zugehörige Komponente merkt.

## 4.5 Komponenten

**Seiten und Seitenfragmente**

Um eine dynamische Seite aufzubauen, verwendet WebObjects – wie im letzten Kapitel ausführlich dargestellt – Komponenten, die Objekte enthalten können, deren Inhalt erst zur Laufzeit bekannt ist. Diese dynamischen Elemente sind in der Lage, sich selbst in Form von HTML

darzustellen. Für Aussehen und Funktionalität gehört die Komponente `WOComponent` somit zu den wichtigsten Klassen. Komponenten können ganze Seiten oder Seitenfragmente repräsentieren. Obwohl wir uns immer auf `WOComponent` beziehen, sind es doch immer Klassen, die von dieser Klasse abgeleitet sind.

**Komponente »Main«**

In einer neuen WebObjects-Applikation ist mit `Main` bereits eine solche abgeleitete Klasse enthalten. Für komplexere Anwendungen sind eine Vielzahl von Komponenten erforderlich, für jede Seite eine. Alle Webseiten innerhalb von WebObjects sind Komponenten und bestehen, wie bereits beschrieben, aus drei Dateien, die HTML, Beschreibung der dynamischen Elemente (».wod«-Datei), sowie Seitenlogik in Form einer Javaklasse enthalten.

**Generierung der HTML-Darstellung**

Die Anfrage eines Browsers löst in der Applikation einen Prozess aus, in dessen Verlauf jedes einzelne dynamische Element der Seite die Nachricht `appendToResponse` erhält. Das Element zieht aus den in der Beschreibungsdatei gebundenen Attributen die notwendigen Parameter, um daraus seine HTML-Darstellung selbst zu generieren. Die Seite kann als Baum von Elementen betrachtet werden, der während der Laufzeit nach und nach dynamisch zusammengebaut wird.

**Actionmethoden**

Neben dem Aussehen verfügen Komponenten häufig auch über eigene Funktionalität. Einigen Elementen können deshalb Actionmethoden zugeordnet werden, die innerhalb der WebObjects-Applikation den Aufruf von Javamethoden auslösen, die nach Durchlaufen einer Logik wiederum eine Komponente zurückliefern.

**Dynamische Elemente**

Einzelne dynamische Bereiche einer Seite werden durch dynamische Elemente repräsentiert, die mit Attributen von Objekten verbunden sind, deren Wert meistens erst zur Laufzeit feststeht. Viele dynamische Elemente haben mehrere mögliche Bindungen. Beispielsweise hat ein Bildbutton (`WOActiveImage`) Bindungen für die URL des Bildes und für die Actionmethode, die er auslösen soll.

**Komponentenklassen**

Komponenten existieren nicht für sich allein, sondern immer mit Bezug zu einer Session und Applikation. Sie werden von diesen übergeordneten Objekten gesteuert. Aus Objektsicht erbt die Java-Klasse einer Komponente von `WOComponent` die vollständige Grundfunktionalität für die Generierung dynamischer Seiten. Damit sind die für einen Anfrage-Antwort-Zyklus notwendigen Methoden `awake`, `takeValuesFromRequest`, `invokeAction`, `appendToResponse` und `sleep` bereits sinnvoll implementiert. Diese werden von der Session oder Applikation zur Beantwortung einer Anfrage, in welche die Komponente involviert ist gerufen.

**pageWithName-Konstruktor**

Neue Instanzen von Komponenten werden nicht etwa über den normalen Java-Konstruktor, sondern über eine spezielle Methode `pageWithName` des WebObjects-Frameworks angelegt. Um in einer Actionmethode die Komponente `Main` zurückzugeben, würde eine solche Methode verwendet werden:

```
public Main goMain {
   return (Main) pageWithName("Main");
}
```

**Main-Typecast**

Die explizite Angabe des Typs (`Main`) ist eigentlich nicht notwendig, soll aber noch einmal verdeutlichen, dass es sich um eine Instanz von `Main.java` handelt. Da diese Klasse, wie alle Komponentenklassen, von `WOComponent` erbt wäre folgende Variante ebenfalls zulässig:

```
public WOComponent goMain {
   return pageWithName("Main");
}
```

**Main-Konstruktor ist unzulässig**

Beide Spielarten sorgen dafür, dass von der zur Komponente `Main` gehörenden Javaklasse `Main.java` ein neues Objekt instantiiert wird. Darüber hinaus werden noch andere interne Vorbereitungen getroffen, weshalb Sie niemals direkt den Klassenkonstruktor verwenden sollten, um Komponenten zu instantiieren. Diese Variante ist *nicht* zulässig:

```
public Main goMain {
   return new Main();
}
```

## 4.6 Frameworks, Bundles, Applikationen

**Frameworks = Module**

In den folgenden Kapiteln werden Sie häufiger auf den Begriff »Framework« stoßen. Ganz allgemein sind Frameworks in der Softwaretechnik abgetrennte Bibliotheken (Module), die bei der Entwicklung verwendet werden können. Sie decken einen definierten Funktionsbereich ab und sind, falls möglich, so aufgebaut, dass sie auch in zukünftigen Projekten verwendet werden können. Durch dieses Merkmal können sie helfen, die Entwicklungskosten bei einem neuen Projekt zu reduzieren.

**Standard-Frameworks**

Auch Apple hat seine standardmäßig mitgelieferten Bausteine in Frameworks organisiert. Die in diesem Kapitel besprochenen Klassen stammen aus dem »WebObjects Framework« (WOF). Im nächsten Kapitel zum Thema Datenbanken konzentrieren wir uns auf das »Enterprise Objects Framework« (EOF).

**Bundles, Applikationen**

In WebObjects wird dieser Mechanismus über so genannte Bundles oder Bündel (wenn es Sie genauer interessiert schauen Sie sich die Klasse `NSBundle` im Paket `webobjects.foundation` an) abgebildet, die primär in zwei Geschmacksrichtungen vorkommen: Frameworks und Applikationen. Ein Framework ist ein »unselbstständiges« Bundle, das von einem anderen Bundle gebunden werden muss, um überhaupt ins Spiel zu kommen. Eine Applikation ist dagegen »selbständig« und wird direkt ausgeführt.

**Bundle = Code + Ressourcen**

Ein Bundle – Framework wie Applikation – kann in WebObjects nicht nur ausführbaren Code enthalten, sondern auch noch andere Ressourcen, etwa Datenmodelle, Webkomponenten, Dokumentation, Konfigurationsdateien, sprachspezifische Dateien. Das im WorkShop vorgestellte Framework `WSEnterpriseObjects` enthält beispielsweise als zusätzliche Ressource das Datenmodell, das WebObjects zur Laufzeit bei allen Datenbankoperationen hinzuzieht. Unsere Frameworks, die mit dem Kürzel »`WI`« (für »WebInterface«) enden, umfassen allesamt als zusätzliche Ressourcen Web-Komponenten, also HTML-Templates und Beschreibungsdateien.

**Trennung und Wiederverwendbarkeit**

In WebObjects werden Frameworks genau aufgrund der eingangs erwähnten Eigenschaften eingesetzt: zum einen können so allgemein gültige und über mehrere Projekte hinweg wieder verwendbare Softwarebibliotheken für WebObjects aufgebaut werden, zum anderen können Funktionsbereiche klar getrennt werden, um die Entwicklungszeit zu minimieren. Bei der internen Änderung einer Klasse muss nur das betroffene Framework neu kompiliert werden.

**Framework: Jar und mehr**

Wir verwenden den Begriff »Framework« ab sofort exklusiv aus WebObjects-Sichtweise. Für uns ist von nun an ein Framework ein Bündel von Ressourcen, das in der Regel eine Jar-Datei und, wenn erforderlich, zusätzliche Ressourcen umfasst. Bedenken Sie aber stets, dass auch eine Applikation ein solches Bündel ist, und somit zusätzliche Ressourcen enthalten kann.

### 4.6.1 Java-Konventionen

Das WebObjects-Paradigma hat nicht zuletzt aufgrund seiner Objective-C-Historie Implikationen, welche die Verwendung der Programmiersprache Java betreffen; etwa auf das Verbot der Strukturierung in Packages, auf Namenskonventionen, die unbedingt zu berücksichtigen sind, oder auf die Lebensdauer der angelegten Instanzen von Java-Objekten.

**Keine Packages**

Java erlaubt die Gruppierung von zusammengehörigen Klassen in Pakete, im Java-Jargon »Packages«. Beispielsweise ist `java.lang` ein solches Paket, das die Standardklassen gruppiert, welche sich auf die Programmiersprache selbst und deren Basistypen beziehen. Weil Komponenten ausschließlich über den Namen instantiiert werden, und weil die Laufzeitumgebung die zugehörige Klasse nur unter den Klassen sucht, die keinem solchen Paket zugeordnet sind, dürfen die Java-Klassen der Komponenten in WebObjects keinesfalls Packages zugeordnet werden.

**Eindeutige Benennung**

Aus der Vorgabe, dass es keine Packages gibt, lässt sich noch eine weitere Anforderung ableiten. Alle Komponenten müssen innerhalb einer Applikation eindeutig benannt sein, nicht nur innerhalb eines Frameworks, oder der Applikation an sich, sondern innerhalb aller in der Applikation verwendeten Frameworks. Wenn in irgendeinem zum Projekt gebundenen Framework, eine identisch benannte Komponente existiert, ist bereits undefiniert, welche zur Laufzeit genommen wird, was zu hässlichen und schwer zu findenden Fehlern führen kann.

**Lebensdauer und Caching**

Wie lange existiert die Instanz einer Komponente? Im Prinzip würde immer nur eine aktive Komponente genügen. Nur die Komponente, welche die Response generiert hat, müsste bis zum nächsten Request aufgehoben werden, bis möglicherweise eine andere Komponente aktiv wird. Da aber die meisten Browser Backtracking erlauben, speichert WebObjects standardmäßig die letzten 30 verwendeten Komponenten in einem Komponentencache. Jede Transaktion hat eine eindeutige Nummer, die Context-ID. Anhand dieser ID werden Komponenteninstanzen im Cache gespeichert und bei Bedarf reaktiviert. Im Falle des Backtrackings wird die Komponente aus dem Cache geholt und mittels `appendToResponse` aufgefordert, erneut eine Response zu generieren. Jede neue Komponente wird in diesem Cache gespeichert, nachdem sie ihre Response generiert hat. Wenn der Cache voll ist, wird das älteste Objekt endgültig gelöscht; der Komponenten-Cache ist also eine »first in, first out«-Warteschlange.

**Komponenten in Request und Response**

Per Definition ist eine Komponente zumindest an zwei Anfrage-Antwort-Zyklen beteiligt: dem, bei dem sie als Antwortkomponente die Response generiert und beim darauf folgenden, wo sie als Anfragekomponente für die Entgegennahme des Requests verantwortlich ist. Wenn der Besucher zwischen zwei Transaktionen einfach das Browserfenster schließt, war die letzte Komponente dennoch nur an einem Zyklus beteiligt.

## 4.7 Struktur von Projekten

Die Projekte haben unter Windows 2000 und MacOS X eine unterschiedliche Struktur. Wir beschreiben hier kurz die Struktur von Applikationen und Frameworks unter Windows 2000 und MacOS X. Denn das Verständnis der Struktur ist wichtig, um später den WorkShop verstehen zu können.

**Windows 2000 und MacOS X**

### 4.7.1 Projektstruktur unter Windows 2000

#### Applikation

Eine neu angelegte Applikation sieht so aus:

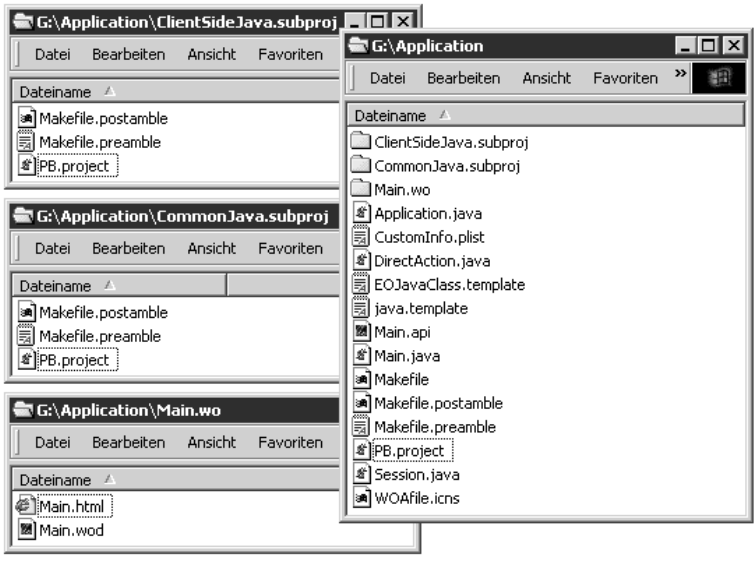


**Abbildung 4.5** Unter Windows 2000 besteht ein WebObhjects-Projekt für eine Applikation aus dem Applikationsordner und drei Unterverzeichnissen für Client Side Java (falls benötigt), Common Java (externer Code) sowie für die Main-Komponente.

In diesem Beispielprojekt finden sich einige typische Dateien und Verzeichnisse:

- `PB.project` ist das eigentliche Projekt. Mit einem Doppelklick öffnen Sie dieses Projekt im Project Builder. **PB.project**
- Die Java-Quelldateien `Application.java`, `DirectAction.java` und `Session.java` sind die Standardklassen, welche die Laufzeitumgebung zum Start der Applikation erwartet. Sie erben von `WOApplication`, `WODirectAction` und `WOSession`. **Java-Quelltexte**

**Main.java** ▶ `Main.java` ist der Quelltext für die Klasse der Komponente »Main«.

**Main.wo** ▶ `Main.wo` enthält Template und Beschreibungsdatei der Komponente »Main«.

**EOJavaClass.template** ▶ `EOJavaClass.template` ist das von Apple vorgesehene Standard-Template für die Generierung von Klassen für Datenbankobjekte. Diese Datei wird vom EOModeler verwendet, sofern sich im selben Verzeichnis ein Datenmodell befindet.

**Makefiles** ▶ Die drei Makefiles sind die von Unix bekannten Vertreter der Buildskripte.

**ClientSideJava.subproj** ▶ `ClientSideJava.subproj` ist ein Unterprojekt, das bei JavaClient-Anwendungen die Ressourcen enthält, die für das Client-Applet bestimmt sind. Wir verwenden dieses Unterprojekt aus den eingangs erwähnten Gründen nicht.

**CommonJava.subproj** ▶ `CommonJava.subproj` ist ein Unterprojekt für externe Java-Dateien. Dieses Projekt benötigen Sie, wenn Sie Jars einbinden möchten.

**Main.api** ▶ `Main.api` ist eine Datei, in die Sie bei Subkomponenten zur Arbeit mit dem WOBuilder die unterstützten Bindings eintragen können.

**CustomInfo.plist** ▶ `CustomInfo.plist` enthält zusätzliche Einträge, die bei der Installation in die Datei `Info.plist` übernommen werden.

**WOAfile.icns** ▶ `WOAfile.icns` ist eine Datei, die das Symbol der Applikation enthält. Für uns ist diese Datei uninteressant.

### Framework

Ein neu angelegtes Framework ist etwas übersichtlicher (siehe Abb. 4.6):

**PB.project** ▶ `PB.project` ist das eigentliche Projekt. Mit einem Doppelklick öffnen Sie diese Datei im Project Builder.

**Makefiles** ▶ Die drei Makefiles sind die von Unix bekannten Vertreter der Buildskripte.

**EOJavaClass.template** ▶ `EOJavaClass.template` ist das von Apple vorgesehene Standard-Template für die Generierung von Klassen für Datenbankobjekte. Diese Datei wird vom EOModeler verwendet, sofern sich im selben Verzeichnis ein Datenmodell verwendet.

**java.template** ▶ `java.template` ist die Datei, die für neue Javaklassen im Projekt als Gerüst verwendet wird.

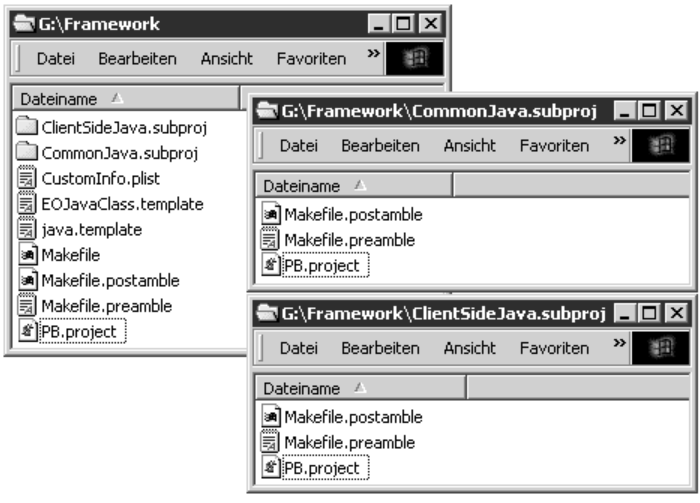


**Abbildung 4.6** Frameworks enthalten unter Windows 2000 zwei Verzeichnisse für den Java-Client und für externen Code.

- `ClientSideJava.subproj` ist ein Unterprojekt, das bei JavaClient-Anwendungen die Ressourcen enthält, die für das Client-Applet bestimmt sind. Wir verwenden dieses Unterprojekt nicht. **ClientSideJava.subproj**
- `CommonJava.subproj` ist ein Unterprojekt für externe Java-Dateien. Dieses Projekt benötigen Sie, wenn Sie Jars einbinden möchten. **CommonJava.subproj**
- `CustomInfo.plist` enthält zusätzliche Einträge, die bei der Installation in die Datei `Info.plist` übernommen werden. **CustomInfo.plist**

In diese Verzeichnisse werden auch alle Dateien kopiert, die Sie dem Projekt hinzufügen.

### 4.7.2 Projektstruktur unter MacOS X

#### Applikation

Eine neu angelegte Applikation sieht aus wie in Abbildung 4.7 gezeigt.

- `Application.pbproj` ist das eigentliche Projekt. Mit einem Doppelklick öffnen Sie dieses Projekt im Project Builder. **Application.pbproj**
- Die Java-Quelldateien `Application.java`, `DirectAction.java` und `Session.java` sind die Standardklassen, welche die Laufzeitumgebung zum Start der Applikation erwartet. Sie erben von `WOApplication`, `WODirectAction` und `WOSession`. **Java-Quelltexte**
- `Main.java` ist der Quelltext für die Klasse der Komponente »Main«. **Main.java**

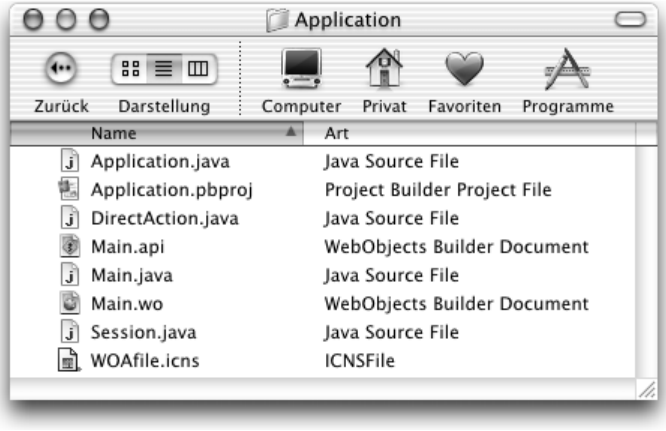


**Abbildung 4.7** Unter MacOS X besteht ein WebObjects-Applikationsprojekt dagegen aus einem Verzeichnis, das unter anderem das Package »Main.wo« enthält.

**Main.wo**

- `Main.wo` ist ein Package und enthält Template und Beschreibungsdatei der Komponente »Main«. Um den Inhalt eines Pakets anzuzeigen klicken Sie mit gedrückter Ctrl-Taste auf den Dateinamen und wählen im Menü »Paketinhalt anzeigen«. Pakete sind Verzeichnisse, die sich wie eine einzelne Datei verhalten.

**Main.api**

- `Main.api` ist eine Datei, in die Sie bei Subkomponenten zur Arbeit mit dem WOBuilder die unterstützten Bindings eintragen können.

**WOAfile.icns**

- `WOAfile.icns` ist eine Datei, die das Symbol der Applikation enthält. Für uns ist diese Datei uninteressant.

### Framework

**Nur eine Projektdatei**

Ein neu angelegtes Framework ist etwas übersichtlicher. Es enthält zunächst nur die Projektdatei, über die sie das Projekt öffnen können (siehe Abb. 4.8).

**Grund: zentrale Stelle**

Im Vergleich zu den Windows-Projekten sind in Projekten unter MaxOS X wesentlich weniger Hilfsdateien enthalten, welche die Compilierung betreffen. Der Grund dafür ist einfach: diese Dateien liegen an einer zentralen Stelle. Die Makefiles finden Sie beispielsweise unter `/Developer/ProjectBuilder Extras/WebObjects Support/`. In unserem WorkShop haben wir diese Dateien in unser Projektverzeichnis kopiert, um daran Änderungen vorzunehmen. Das Projekt muss dann natürlich entsprechend angepasst werden, damit es diese Makefiles verwendet.

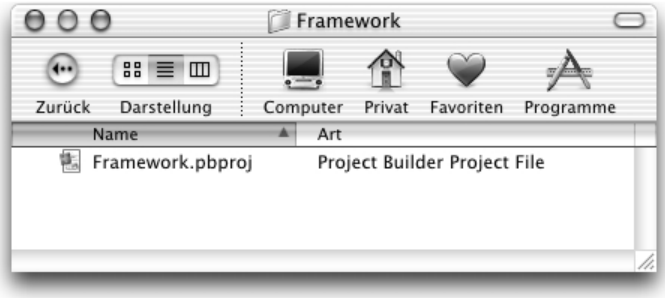


**Abbildung 4.8** Ein WebObjects-Framework unter MacOS X enthält zunächst nur die Projektdatei.

### 4.7.3 Projektstruktur des WorkShops

Beachten Sie bitte, dass unsere Beispielprojekte sowohl die Dateien für MacOS X als auch für Windows 2000 enthalten. Lassen Sie sich dadurch nicht verwirren. Auf Ihrer Plattform werden auch nur die dafür benötigten Dateien verwendet. Das Verzeichnis für das Projekt mit den Datenbank-Klassen unseres WorkShops sieht beispielsweise so aus:

**Beispielprojekt**

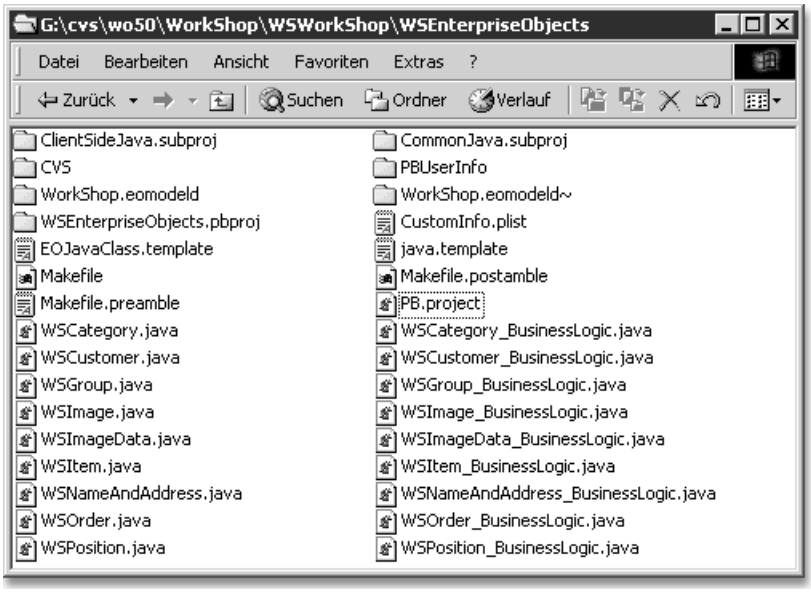


**Abbildung 4.9** Bei unserem später im Dateil vorgestellten »WorkShop« sind jeweils alle Dateien für Windows 2000 und MacOS X enhalten.

**WorkShop-Dateien**

Hier sind nun schon wesentlich mehr Dateien enthalten. `WorkShop.eomodeld` ist unser Datenmodell. `PB.project` ist die Projektddatei für Windows 2000 und `WSEnterpriseObjects.pbproj` ist die Projektdatei für MacOS X, die hier als Ordner dargestellt wird, weil sie unter MacOS X ein Verzeichnis ist, das als eine Datei erscheint, was als Paket bezeichnet wird. Das CVS-Verzeichnis stammt von unserer Versionsverwaltung. Die Datei mit der Tilde ist das Backup der Modelldatei, welches der EOModeler nach jeder Änderung anlegt.

### 4.7.4 Struktur installierter Projekte

Wenn eine Applikation oder ein Framework installiert wird, hat es ebenfalls eine klare Struktur.

#### Applikation

Eine installierte Applikation sieht so aus. Die Installation ist unter Windows und MacOS X identisch, wir zeigen hier die Abbildung von MacOS X:

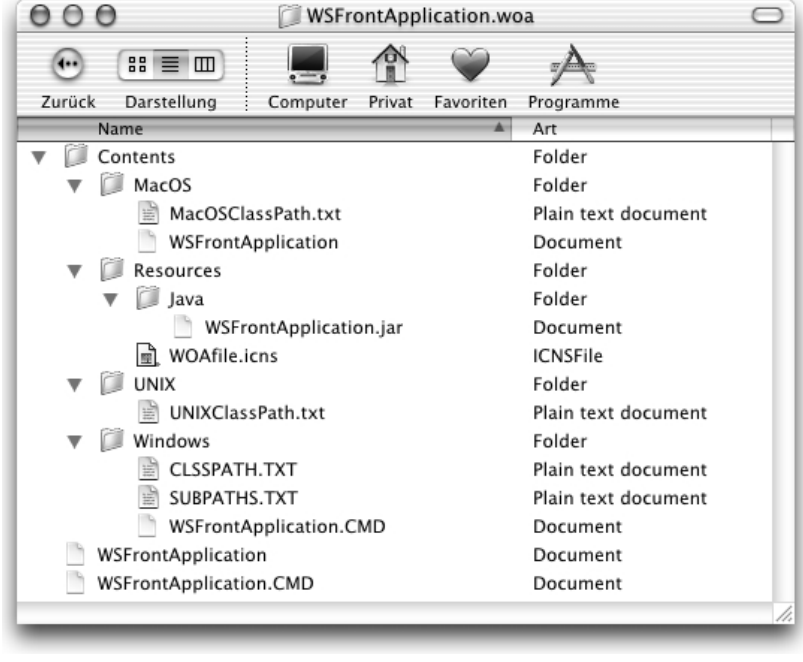


**Abbildung 4.10** Die installierte Applikation »WSFrontApplication« aus unserem »WorkShop« enthält auch die Ressourcen für die verschiedenen Betriebssysteme.

Es gibt ganz unten zwei Haupt-Startskripte, die wohlgemerkt keine Applikationen sind, sondern Shell- bzw. Kommandozeilenskripte. Die ohne Dateierweiterung ist für Unix (MacOS X, Solaris, Linux) bestimmt. Die Datei mit der Erweiterung »CMD« ist für Windows. Es existiert ein Verzeichnis »Resources«, welches Inhalte für alle Plattformen enthält, wie Jar-Datei mit den Javaklassen oder die Icon-Datei, die für uns nicht wichtig ist. In den Verzeichnissen der jeweiligen Betriebssysteme liegen dann nur noch die für diese Plattform notwendigen Anpassungen.

**Applikationsdateien**

### Framework

Ein Framework sieht ein wenig anders aus.

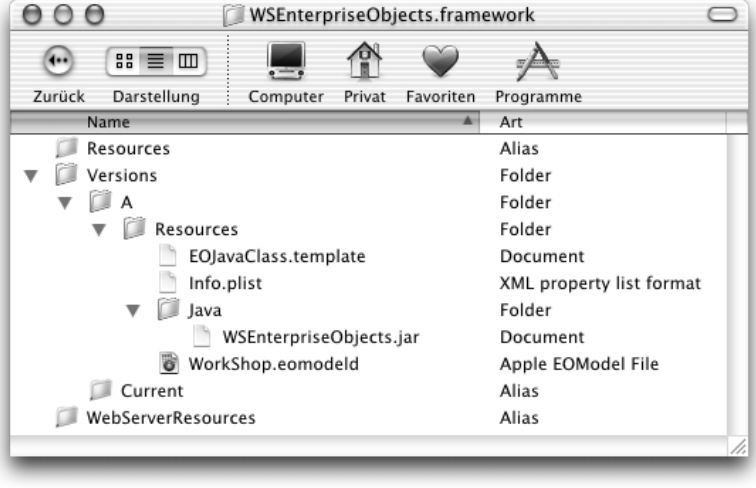


**Abbildung 4.11** MacOS X erlaubt auch die Versionierung von Frameworks.

Hier gibt es unter MacOS X noch eine Versionierung, wodurch ein bestimmtes Framework gleichzeitig in verschiedenen Versionen installiert werden kann. Applikationen und andere Frameworks können so eine ganz bestimmte Version eines Frameworks einbinden, was beispielsweise notwendig ist, wenn ein neueres Framework mit einem älteren inkompatibel ist. Unter Windows ist diese Versionierung standardmäßig nicht aktiviert, aber ebenfalls möglich.

**Versionierung unter MacOS X**

Wir werden die Versionierung nicht erklären, da sie für uns unerheblich ist. »Resources« ist ein Verweis auf »Resources« der aktuellen Version, und »WebServerResources« ist analog ein Verweis auf die WebServer-Ressourcen der aktuellen Version. In Versions gibt es einen Verweis »Current« der nochmals auf die aktuelle Version, bei uns »A«, verweist.

**Versionierungsdateien**

In »A« liegt schließlich das Framework, wie es ohne Versionierung aussehen würde. In »Resources« liegen alle Inhalte des Frameworks, hier eine Datei »Info.plist«, mit zusätzlichen Informationen, die zur Laufzeit verwendet werden und unser Datenmodell »WorkShop.eomodeld«. In einem eigenen Verzeichnis »Java« liegt schließlich die Jar-Datei mit allen Klassen dieses Frameworks.

**Datenbankklassen-Template**

Warum das Template für die Datenbankklassen mit installiert wird, weiß wohl allein Apple. Es müsste da nicht liegen.

### 4.7.5 Standard-Objekte in neuen Projekten

**Vererbung**

Um das objektorientierte Design voll auszuschöpfen unterstützt WebObjects Vererbung. Wenn Sie in ProjectBuilder eine neue WebObjects-Applikation anlegen, werden darin bereits vier vererbte Klassen standardmäßig angelegt:

**Applikation**

- Die Klasse `Application.java` erbt von `WOApplication`. Hier können Sie Ihre Anpassungen der Applikation vornehmen. Die WebObjects-Laufzeit erzeugt standardmäßig eine Instanz dieser Klasse.

**Session**

- Die Klasse `Session.java` erbt von `WOSession`. In dieser Klasse können Sie das Session-Objekt an Ihre Erfordernisse anpassen. Die Standardimplementierung von `WOApplication` erzeugt ein solches Objekt. Falls Ihre Sessionklasse nicht `Session` heißt, müssen Sie die Methode `createSessionForRequest` in der Applikation so überschreiben, dass Ihre Sessionklasse instantiiert wird.

**DirectAction**

- Sessions haben den Nachteil, dass sie einen großen organisatorischen Overhead mit sich bringen. Neben einem Session-Objekt wird daher auch eine Direct Action angelegt, die von `WODirectAction` erbt. Eine Direct Action ermöglicht einen sessionlosen Request. Da Direct Actions keine Session haben, enthalten ihre URLs auch keine Session-IDs und sind dadurch vollständig statisch. Dadurch können von Direct Action URLs im Gegensatz zu normalen Komponenten-URLs Bookmarks im Browser angelegt werden.

**Komponenten**

- Alle Komponentenklassen erben von `WOComponent`. Sie verfügen somit wie die Applikation und die Session über alle an dem Request-Response-Zyklus beteiligten Methoden: `takeValuesFromRequest`, `invokeAction` und `appendToResponse`. In einer neu angelegten Applikation ist bereits eine leere Komponente mit der zugehörigen Javaklasse `Main.java` vorhanden.