

Teil I: Einleitung und Basistechnologien

Kap1: Einleitung

Plattformspezifische Voraussetzungen: Java EE mit Enterprise JavaBeans (EJB) für den Anwendungskern sowie Webkomponenten (Servlet und JSP) und Webframework Struts für die Webschicht (unter Benutzung verschiedener Basistechnologien).

Modellierungstechnische Voraussetzungen: Softwarearchitekturen für Webapplikationen sowie UML-Aktivitäten zur Modellierung funktionaler Abläufe.

Methodische Entwicklung von Webapplikationen: Anforderungsermittlung -> Software-spezifikation -> Entwurfsklassenmodell für Struts-basierte Webschicht und EJB-basierten Anwendungskern.

Kap2: Basistechnologien

Webapplikationen und Webserver

Webapplikation: HTTP-basierte Client -Server-Anwendung

Client (HTML-Browser): übernimmt Oberfläche der Benutzungsschnittstelle, stellt Anfragen an Server und erhält dessen Antworten.

Server: restliche Benutzungsschnittstelle, Anwendungskern, Datenhaltung.

Unterschiede zu **Desktop-Applikationen:**

- ◆ *Nachteile:* weniger Gestaltungsmöglichkeiten UI, nur 1 Fenster, Programmabläufe schlechter kontrollierbar, erhöhter Kommunikationsaufwand
- ◆ *Vorteile* bei Installation, Wartung, Portabilität und Verfügbarkeit

Rich Clients: nicht nur UI, auch Datenverarbeitung.

Code on Demand: Programme oder einzelne Funktionen vom Server laden und clientseitig ausführen (Java Applets, JavaScript) -> erweiterte Benutzerinteraktion.

Einfache **Webserver** (Apache HTTPD, Microsoft IIS) liefern als Dateien abgelegte HTML-Dokumente und Bilder an Client aus. Weitergehende Anwendungen über modulare Erweiterungen oder spezielle Webserver.

URI

Uniform Ressource Identifier (URI) ist weltweit gültiger und eindeutiger Bezeichner für **Ressource**. Webseiten sind Darstellungen von auf Server liegenden Ressourcen.

Ressourcen können unterschiedliche Form und Darstellungen haben.

Aufbau: **Schema der URI** ist protokollabhängig.

scheme „:“ <*scheme-specific-part*>

http_URL: „http“ „/“ host [„:“ port] [*abs_path* [„?“ query]], zB

<http://www.fernuni-hagen.de>

<mailto:Kurs1794@FernUni-Hagen.de?subject=Klausuranmeldung>

Varianten:

Uniform Resource Locator (URL): identifiziert Ressource eindeutig über Ortsangabe

Uniform Resource Name (URN): identifiziert Ressource eindeutig über Namen (kein Einsatz bei HTTP)

Protokolle

- ◆ **Vermittlungsschicht: Internet Protocol (IP)** ist paketorientiert, unzuverlässig und verbindungslos
- ◆ **Übertragungsschicht: Transmission Control Protocol (TCP)** kann durch Ports und IP-Adressen beidseitige Verbindung herstellen. Datenüberprüfung und Fehlerkorrektur. **Domain Name Service (DNS)**: ordnet IP-Adressen Namen zu.
- ◆ **Anwendungsschicht: Hypertext Transfer Protocol (HTTP)** stützt sich auf TCP/IP ab. Message-basierte Pull-Technologie: Client-Request zieht Server-Response. HTTP-Nachricht als einfacher ASCII-Text formatiert. **GET-Übertragungsmethode**: Clientdaten (als Parameterangaben im Query-String) in der URL übertragen. **POST-Übertragungsmethode**: Parameter im Anschluss an Header gesendet, für umfangreiche (sicherheitsrelevante) Daten. **HTTPS**: durch Einsatz eines **Secure Socket Layer (SSL)** transparent verschlüsselte Datenübertragung. **Zustandslosigkeit HTTP**: Zustand oft nicht benötigt, man erspart sich aufwendiges Recovery bei Übertragungsfehlern. Abhilfe durch ständig zirkulierende Session-ID.

Dokumentenformate

MIME (Multipurpose Internet Mail Extension): Definition der Inhalts- und Medientypen sowie Codierung der Daten.

HTML mit Referenzen, Tags für Formateigenschaften, Kommentaren, Sonderzeichen, Formularen. **XHTML**: Syntax von HTML auf Basis XML definiert.

XML: Metasprache zur Beschreibung hierarchischer (Dokument-) Strukturen und Definition von Auszeichnungssprachen (zB XHTML). Trennung von Struktur und Inhalt sowie Plattformunabhängigkeit. Baumartig aufgebaute Dokumente mit Elementen und deren Attributen. Wenn durch **DTD** bzw **Schema** definiert -> gültig, sonst -> wohlgeformt.

Inhalt

- ◆ **Statischer Inhalt**: auf dem Webserver abgelegte Dateien (HTML-Dokumente, Bilder)

- ◆ **Dynamisch erzeugter Inhalt:** als Antwort auf Anfrage von Applikation erzeugt. zB **Common Gateway Interface (CGI)** als Schnittstelle zwischen Webserver und weiteren Programmen. Aber -> hoher serverseitiger Ressourcenverbrauch, da für jede Anfrage eigener Prozess gestartet.
- ◆ **Dynamischer Inhalt:** HTML + Code-On-Demand-Techniken (=DHTML)

Session

HTTP ist zustandslos, Webapplikationen idR nicht. Zustände beziehen sich oft auf jeweiligen Benutzer und werden meist auf Serverseite gespeichert. IP-Adresse reicht zur Client-Identifikation nicht aus -> dynamische IP's | mehrere Rechner hinter Router mit nur einer IP-Adresse. Problemlösung:

Konzept der **Sitzung (Session)**: logische, dauerhafte Verbindung zwischen Client und Server. **Session-ID** vom Server erzeugt und Client bei jeder Antwort übermittelt (Ausnahme Cookies). Dieser schickt dann bei jeder Anfrage die Session-ID mit. Sitzungsende durch explizites Abmelden oder Zeitüberschreitung.

3 Möglichkeiten, **Session-ID** in jede Anfrage einzubinden:

- ◆ **URL-Rewriting:** Session-ID als zusätzlicher Parameter in URL.
- ◆ **Versteckte Formularfelder:** `<input type="hidden" name="session-id" value="1234567">`
- ◆ **Cookies:** vermeiden, dass jedes Formular/jede URL modifiziert werden muss. Werden mit jeder Anfrage an Server geschickt. Müssen aber vom Client akzeptiert werden.

Teil II: Webschicht der Java Plattform (JEE 5)

Kap3: Einstieg

Java EE im Überblick

Die Spezifikation gibt einen Rahmen für die Entwicklung verteilter, mehrschichtiger Anwendungen mit modularen Komponenten vor. Besteht aus **vier** auf der J2SE basierenden **Architekturkomponenten (Containern)**, die als Laufzeitumgebung und Anbieter von Standardservices fungieren. Interaktion der Applikationskomponenten erfolgt ausschließlich über Methoden und Protokolle der Container.

- ◆ **EJB-Container:** läuft auf Server. Konzepte zur Umsetzung des Anwendungskerns
- ◆ **Webcontainer:** läuft auf Server. Webkomponenten Servlet und JSP.
- ◆ **Applet Container:** läuft auf Client (Java-Plugin für Browser). GUI wie Desktop-Applikationen, Kommunikation mit Webkomponenten via HTTP.
- ◆ **Application Client Container:** läuft auf Client. Klassische Java-GUI-Applikationen. Kommunikation mit EJB (RMI) und Webkomponenten (HTTP).

Verzeichnisstruktur Webschicht

WAR (Web Archive): gewöhnliche Zip-Datei, die Verzeichnisstruktur in gepackter Form enthält. **Deployment:** Installieren, Konfigurieren, Ausführen von JEE-Webapplikationen.

Verzeichnisse:

/Name(| / | /WEB-INF(| /classes/ | /lib/*.jar | web.xml) | /META-INF/) =>*

Wurzelverzeichnis(| *direkt erreichbar* | Webschicht(| Javaklassen | Java-Bibliotheken | Konfigurationsdatei) | Metadaten über WAR)

Deployment-Descriptor web.xml: zB

- ◆ Servlet Deklaration: Name, Klassenname (incl. Paketstruktur)
- ◆ Servlet Mapping: Servletname, URL-Pattern
- ◆ Welcome File List: Startseite

Kap4: Servlet

Request – Ablauf

- ◆ Browser -> Webcontainer -> Servlet -> Anwendungskern -> Servlet -> Webcontainer -> Browser.
- ◆ HTTP-Request -> `HttpServletRequest`(Request | Response -> `PrintWriter`)
- ◆ `create`, `service()`, `doPost()`, `getParameter()`, `<<buisenessMethod>>()`, `setContentType()`, `getWriter()`, `println()`, `destroy`

Aufgabe, Lebenszyklus, Thread-Sicherheit

Webkomponenten und insbesondere Servlets stellen eine Art Zwischenebene zwischen Clients und Anwendung auf dem Server dar.

Mögliche (nicht unbedingt sinnvolle) **Aufgaben Servlet:**

- ◆ vom Client gesendete Daten lesen
- ◆ vom Client gesendete HTTP-Request Header lesen
- ◆ Ergebnisse erzeugen
- ◆ HTTP-Response Header setzen
- ◆ Antwortdokument erstellen

Servlet hat fest definierten **Lebenszyklus** (über Callback-Methoden realisiert): 3 Phasen

- ◆ Initialisierung
- ◆ Requestbearbeitung
- ◆ Zerstörung

Webcontainer startet für Dauer einer Applikation genau eine Instanz pro Servlet. Pro Request 1 Thread, evtl gleichzeitige Zugriffe -> Servlet muss **thread-sicher** sein.

3 Möglichkeiten:

- ◆ **keine Instanzvariablen**, ausschließlich (Methoden-)lokale Variablen - diese sind stets Thread-lokal. Alle Variable als Parameter übergeben.
- ◆ alle Instanzvariablen in der init()-Methode des Servlets mit Werten füllen und (abgesehen von destroy()) **nur noch lesend zugreifen** - es können keine Synchronisierungsprobleme auftreten
- ◆ Lese- und Schreibvorgänge zulassen aber mittels **synchronized()** synchronisieren. Nicht sehr zweckmäßig, einzelne Anfragen sollten Servlet-Zustand nicht ändern.

Scopes

Webkomponenten besitzen die Möglichkeit, mit anderen Webkomponenten Daten auszutauschen. Die Servlet-Spezifikation kennt **4** als Interfaces definierte **Scopes** (Behälter mit definierter Lebensdauer zum Ablegen temporärer Daten):

- ◆ **Request-Scope**: HttpServletRequest-Object existiert für Dauer eines Request-Durchlaufs.
- ◆ **Session-Scope**: HttpSession-Object existiert für Dauer einer Session. Über Request-Object mittels getSession() geholt.
- ◆ **Application-Scope**: ServletContext-Object existiert für Dauer der gesamten Webapplikation. Für Webapplikation global gültige Daten, beim Beenden der Applikation gelöscht. In Servlet mittels getServletContext() geholt.
- ◆ **Page-Scope**: JSP-lokale Daten.

JavaBeans

Werden (neben Strings sowie Wrappern für Standarddatentypen) zum Ablegen der Daten in den Scopes verwendet. Besitzen keine öffentlichen Attribute, dafür **öffentliche Schnittstelle** in Form Accessor(getter/setter)-Methoden. **Getter** sind parameterlos (haben Rückgabewert), **Setter** haben genau einen Parameter vom Typ der Property. Getter/Setter zu Array-Properties (indexed Properties) dürfen int-Parameter beziehen. JSP können mittels Expression Language (EL) Attributwerte einer JavaBean-Instanz beziehen.

Servlet API

Komfortable Programmierschnittstelle für Java EE Webapplikationen.

<i>Interface</i>	<i>Methoden</i>
HttpServletRequest (<i>abstrakte Klasse</i>)	doGet, doPost, init, destroy, getServletContext
HttpServletRequest	getParameterNames, getParameter, getParameterValues, getParameterMap getHeaderNames, getHeader, getHeaders getRequestURI, getRequestURL getSession, getRequestDispatcher

<i>Interface</i>	<i>Methoden</i>
HttpServletResponse	setContentType, getWriter, encodeURL, addCookie
HttpSession	invalidate, getServletContext
ServletContext	getRequestDispatcher
RequestDispatcher	forward, include

Jedes vom Entwickler zu implementierende Servlet wird von der abstrakten Klasse *HttpServlet* abgeleitet.

Kap5: Java Server Page

Request-Ablauf

Die Aufgabe einer JSP besteht darin, zum Abschluß des Requestablaufs die Ausgabe an den Client zu übernehmen. In JSP wird statischer Inhalt im Format des Ausgabedokuments (HTML, XML, ...) angegeben, zur Einbettung von dynamisch erzeugten Inhalt existieren verschiedene Techniken. JSP sind textbasierte Dokumente, welche die Arbeitsteilung zwischen Entwicklern und Webdesignern fördern.

JSP werden vom Webcontainer beim ersten Request transparent in Servlet übersetzt, welches bei allen folgenden Requests aufgerufen wird.

Das (Controller-) Servlet leitet mit der forward()-Methode des IF RequestDispatcher an die JSP weiter, damit diese die Gestaltung der Präsentation übernimmt. Vorher werden die von der JSP benötigten Daten (zB Ergebnisse von Anwendungskernaufrufen) mittels setAttribute() im Request-Scope abgelegt und über das Request-Objekt ein Request-Dispatcher-Objekt angefordert. Die JSP holt sich die Daten mittels getAttribute().

JSP-Konstrukte

- ◆ **Template-Text:** jede Form von statischen Text, der an Client gesendet wird. JSP mit ausschliesslich Template-Text sieht aus wie normales HTML-Dokument.
- ◆ **JSP-Kommentar:** bleibt beim Übersetzen JSP unberücksichtigt und wird - im Gegensatz zu HTML-Kommentaren- nicht mit Response an Client geschickt. Nur für die Entwickler. `<!--Kommentar-->`
- ◆ **JSP-Direktive:** Informationen für Web Container zur Verarbeitung der Seite: import-Anweisungen, Ressourcen einbinden, Tag Libraries deklarieren.
`<%@ directive attribute1="value1" attribute2="value2 ... %>`
directive aus {page, include, taglib}
attribute aus {(import, errorPage...) | file | (prefix, uri)}.
- ◆ **Scripting-Elemente:** ermöglichen JSP über dynamisch erzeugten Inhalt zu verfügen.
 - **JSP-Deklaration:** Vorsicht Thread-Sicherheit! -> siehe Servlet.
`<%! private int ergebnis; %>`
 - **JSP-Ausdruck:** auswertbarer Java-Ausdruck, wird in dem an Client gesendeten Ergebnis-Dokument durch seinen Wert ersetzt.

`<%= (new java.util.Date()).toGMTString() %>`

- **JSP-Scriptlet:** Java-Anweisung für Berechnungen, DB-Zugriffe, Anwendungskernaufrufe. Für Client sichtbare Ausgabe nur, wenn explizit veranlasst (-> `println()`).

`<% if (Math.random() < 0.5) {out.println(„gewonnen!“)} %>`

- ◆ **JSP-Aktionen:** decken Teilmenge der Funktionalität von Scripting-Elementen ab, allerdings mit für Webdesigner zugänglicherer XML-Syntax.

`<jsp:action attribute1=“value1“ attribute2=“value2“ ... />`

action aus {forward, include, useBean, setProperty, getProperty...}

Expression Language (EL)

Scripting Elemente und JSP-Aktionen sind aus SE-Sicht kritisch zu hinterfragen und am besten gar nicht zu verwenden. Schränkt man eine JSP sinnvollerweise dahingehend ein, dass sie nur die **Präsentation** übernimmt und über **HTML** hinausgehende Techniken ausschliesslich zur Darstellung von **dynamisch erzeugten Inhalt** einsetzt, muss sie im Grunde lediglich die in einem Scope abgelegten Ergebnisse von Aufrufen des Anwendungskerns auslesen können. Dies unterstützen EL und JSTL.

Mit EL kann man von einer JSP aus auf in Scope gespeicherte Objekte und (sofern es JavaBeans sind) ihre Properties zugreifen. EL verfügt über relationale, logische und arithmetische Operatoren und implizite Objekte. Sie läßt sich um „Funktionen“ erweitern. Ihre Syntax ist einfach und der Funktionsumfang gezielt eingeschränkt (Objekte können nicht ohne weiteres erzeugt oder modifiziert werden).

EL-Ausdruck: `#{<Ausdruck>}` bzw `#{<Ausdruck>}` für verzögerte Ausführung bei JSF.

Zugriff auf Objekte: `#{kunde.firstName}` oder `#{kunde[„firstName“]}`

Geschachtelte Objekten: `#{kunde.adresse.strasse}`, `#{kunde[„adresse“][„strasse“]}`.

Maps: über Wert des Schlüssels.

Arrays und Listen: nur Variante mit eckigen Klammern.

Implizite Fehlerbehandlung: wenn angeforderte Objekte *null* sind oder Indexgrenzen überschritten wurden, wird einfach *null* zurückgeliefert (keine Fehlermeldung). Erspart Abfragen aller Objekte und Attribute auf *null*.

Operatoren: die üblichen relationalen, logischen und arithmetischen Operatoren sowie ihre textuellen Alternativen (zB `==` und `eq`). EL nimmt implizite Typkonvertierungen vor, sofern möglich.

Implizite Objekte: ohne explizite Erzeugung immer zugreifbar.

pageContext, (page|request|session|application)Scope, initParam, param, paramValues, header, headerValues, cookie.

Java Standard Tag Library (JSLT)

Zur Erweiterung der Ausdrucksmächtigkeit der EL kann (zusätzlich) die **JSTL** verwendet werden. Ihre Tags erweitern die JSP-Aktionen und erlauben **Operationen** wie **Iterationen** über Objektmenge, **bedingte Auswertungen** oder spezielle **Formatierungen** der Ausgabe.

JSTL ist benutzerdefinierte Tag-Library mit 5 Bibliotheken:

core | xml | fmt | sql | functions

Bibliotheken müssen über taglib-Direktive am Anfang JSP eingebunden werden (mit uri und prefix). In **core-Bibliothek** finden sich u.a.:

- ◆ **url-Tag:** für URL-rewriting
- ◆ **if-Tag:** für bedingte Auswertungen.
- ◆ **choose - when - otherwise:** simuliert if-then-else Anweisung
- ◆ **forEach-Tag** (Attribute begin, end, var): Schleife für Objektmengen, Array, Liste oder Map.

In **Function-Tag-Library** finden sich Funktionen, die im Wesentlichen zur String-Bearbeitung gedacht sind. Es können auch **eigene (benutzerdefinierte) Bibliotheken** definiert werden.

Kap6: Servlets und JSP aus Software Engineering Sicht

SE-Grundregeln

Separation of Concerns: möglichst wenig Überlappung der Aufgaben von Softwarekomponenten

Kohäsionskriterium: jede Softwarekomponente für eine fachlich/logisch zusammengehörende Aufgabe zuständig.

MVC-Pattern: grundlegende Strukturierungsvorgabe für interaktive Systeme

- ◆ **Model:** Anwendungslogik und Anwendungsobjekte [*Anwendungskern*]
- ◆ **View:** Präsentation [*Teil der Benutzungsschnittstelle*]
- ◆ **Controller:** steuert den Ablauf der Interaktionen [*Teil der Benutzungsschnittstelle*]

Zuschnitt von Verantwortlichkeiten

- ◆ **Servlet:** weder Präsentationsaufgaben noch Anwendungslogik. Servlet interpretiert eingehenden Request, setzt dabei Aufrufe an Anwendungskern ab und leitet schließlich an JSP weiter (zur Ausgabe)
- ◆ **JSP:** weder Anwendungskern aufrufen noch Anwendungslogik selbst implementieren. Nur Präsentation, über HTML hinausgehend nur Darstellung von dynamisch erzeugten Inhalt. EL und JSTL genügen zum Auslesen, Scripting-Elemente und bestimmte JSP-Aktionen nicht benutzen.
- ◆ **Anwendungskern:** implementiert Anwendungslogik und Anwendungsobjekte. Reiner Dienstleister, reagiert nur auf Aufrufe von (ihm natürlich unbekanntem) Servlets.

Vorteile der Arbeitsteilung: fördert Komplexitätsbeherrschung, Aufteilung von Tätigkeiten unter Entwicklern, Test und Fehlersuche vereinfacht, Wartbarkeit verbessert.

KE 3&4

Teil III: Softwarearchitektur

Kap7: Motivation und Grundlagen

Die **Softwarearchitektur** legt das grundsätzliche Strukturierungsschema der Software und das Zusammenspiel der strukturbildenden Bestandteile (**Architekturkomponenten**) fest. Diese bilden eine **Zerlegung**: jedem Software-Element ist genau eine Architekturkomponente zugeordnet. Die **Architektur** besitzt somit *globalen Charakter*, während der **Softwareentwurf** innerhalb des vorgegebenen Rahmen *lokale Realisierungsentscheidungen* trifft.

Wahl der Architektur: überwiegend von **nicht-funktionalen Anforderungen** bestimmt.

- ◆ aus *Benutzersicht*: zB Performanz, Sicherheit, Verfügbarkeit
- ◆ aus *technischer Sicht*: zB Testbarkeit, Integrierbarkeit, Wartbarkeit

Eine Architektur stellt immer einen Kompromiß dar. Eine **gute Architektur** erfüllt mindestens die folgenden **Kriterien**:

- ◆ adäquate Basis zur Realisierung der funktionalen und nicht-funktionalen Anforderungen
- ◆ möglichst unabhängig von den Spezifika der Anwendung
- ◆ abgegrenzte Aufgabenbereiche mit hoher Kohäsion (SoC)
- ◆ präzise festgelegtes Zusammenspiel der Architekturkomponenten: einfache und überschaubare use-dependencies (möglichst keine Zyklen)

Gute Architekturen fördern Wiederverwendbarkeit, Komplexitätsbeherrschung, Wartbarkeit und Änderbarkeit, sowie unabhängige Entwicklung von Architekturkomponenten durch Spezialisten.

Beispiel: **3-Schichten-Architektur** (Globalitätseigenschaft, antizyklische von oben nach unten verlaufende Benutzungsabhängigkeiten).

Kap8: Architekturmuster für Webapplikationen

Client-Server-Architektur

Softwaresystem in 2 Architekturkomponenten mit klar getrennten Aufgabenbereichen aufgeteilt: **Server** (bietet Dienste an) und **Client** (nutzt die Dienste) -> gutes Architekturmuster.

Ist **kanonisches Architekturmuster für Webapplikationen**: Browser zeigt UI-Oberfläche, Server hat restliche UI und Anwendungskern inklusive persistenter Datenhaltung. HTTP als Kommunikationsprotokoll (Pull-Technologie). Sehr gute **Portierbarkeit** (Browser sind allgegenwärtig), erhöhte **Sicherheit** bei geringerem Aufwand durch Konzentration der Sicherheitsmassnahmen auf den Server.

Schichtenarchitektur

Verfeinerung der Client-Server-Architektur, die **Server** in iterierter Weise wieder als **Client-Server-Architektur** strukturiert. Architekturkomponenten aufgrund der **einseitig gerichteten Benutzungsabhängigkeiten** als übereinander liegende Schichten dargestellt. **Strikte Schichtenarchitekturen** erlauben nur Zugriff auf nächstniedrigere Schicht. **Aufrufzyklen** sind ausgeschlossen, jede Schicht kennt nur eine andere Schicht, die ihre Dienste vollkommen **transparent** anbietet. -> ein gutes Architekturmuster.

- ◆ **3-Schichten-Architektur**: externe Schnittstelle -> Anwendungskern -> persistente Datenhaltung.
- ◆ **5-Schichten-Architektur für Webapplikationen**: das eigenständige komplexe Softwaresystem **Server** wird innerhalb der Client-Server-Architektur mit einer 4-Schichten-Architektur strukturiert, was zusammen mit dem Client 5 Schichten ergibt. -> ein gutes Architekturmuster.
 - **Persistente Datenhaltung**: meist relationale DB
 - **Anwendungsobjekte (Entitäten)**: das auf Entitätsklassen und ihre Beziehungen reduzierte Klassenmodell der Webapplikation.
 - **Anwendungslogik**: die fachliche Funktionalität der Webapplikation.
 - **Webschicht des Servers**: Entgegennahme von Clientanfragen, Aufrufen der Anwendungslogik, Absenden der Serverantworten (Servlets, JSP, Webcontainer).
 - **Client**: Browser

<i>Client</i>	Browser
<i>Server</i>	Webschicht
	Anwendungslogik
	Anwendungsobjekte
	Datenhaltung

Der **Client** benutzt ausschliesslich die Webschicht. Die **Webschicht** greift nur auf Anwendungslogik und nur in Ausnahmefällen auf Anwendungsobjekte zu. Die **Anwendungslogik** benutzt nur Anwendungsobjekte und Datenhaltung, **Anwendungsobjekte** greifen nur auf Datenhaltung zu.

Vielfach wird zwischen Anwendungsobjekten und Datenhaltung noch eine **Vermittlungsschicht (Middleware)** eingebracht, um die konzeptionelle und technische Kluft zwischen OO-Anwendungsobjekten und relationalen DB zu überbrücken. Ein solches **Persistenzframework** macht die Datenhaltung für darüberliegende Schichten transparent.

Die im Folgenden besprochenen Architekturmuster sind auf technologische Spezifika von Java EE zugeschnitten.

Model-1-Architektur

Besteht aus **Browser**, **Webkomponente** (JSP: Darstellung UND Steuerung-> Scripting Elemente) und **Model** (Anwendungskern). Request an JSP -> Prüfungen und Modellaufrufe -> Response an Browser. **Problem:** mangelnde Kohäsion der nur über JSP realisierten Webkomponente. Keine Trennung Darstellung/Steuerung, Vermischung von HTML und Java-Code. -> kein gutes Architekturmuster.

Model-2-Architektur

MVC-Pattern auf Java EE - Webapplikationen angewendet. **C** nimmt User Event entgegen, ruft **M** zur Umsetzung Benutzerwunsch auf, ruft **V** zur Bildschirmdarstellung des Ergebnisses auf -> **C** nimmt User Event entgegen und kontrolliert den zugehörigen Ablauf.

Request -> **C** -> **M** -> **C** legt Ergebnisse in Scope -> **V** (JSP) liest aus Scope -> Response. -> gutes Architekturmodell. In Praxis beachten Entwickler die strenge Arbeitsteilung zwischen Servlet und JSP zu wenig.

Eine **Webapplikation** mit vielen Webseiten kann sehr viele JSP's (≥ 1 JSP pro Webseite) und mehrere Servlets haben. Das **Webframework Struts** arbeitet mit dem **Front Controller Pattern**, bei dem jeder Request von ein und demselben Servlet entgegengenommen wird. Die vom jeweiligen Request geforderte individuelle Funktionalität wird durch weitere zum Controller gehörende Klassen realisiert, die vom Servlet entsprechend aufgerufen werden.

Teil IV: Webframework Struts

Kap9: Struts im Überblick

Motivation

Webframeworks nehmen Entwicklern viel technisch anspruchsvolle low-level Programmierarbeit bei der Implementierung der Webschicht ab. **Struts** basiert auf **Model-2-Architektur** und übernimmt weitgehend die Umsetzung des **Controllers**, der nur noch in Teilen an die spezielle Anwendung angepasst werden muss.

Definition: „A framework is a reusable design of all or part of a system that is represented by a set of abstract classes and the way their instances interact“

Applikationsspezifische Klassen sind Spezialisierungen bzw Implementierungen von (abstrakten) Klassen des Frameworks bzw von Interfaces des Frameworks. Im Gegensatz zur **Klassenbibliothek** legt **Framework** die Architektur einer Anwendung fest und es herrscht „**Inversion of Control**“ (Framework benutzt Applikationsklassen und nicht umgekehrt).

Struts-Komponenten

- ◆ **Front Controller:** Struts implementiert Front Controller Pattern. Es existiert nur ein einziges Servlet (**ActionServlet**), das alle Requests (bzw alle vom Webcontainer zu **HttpServletRequest-Objekten** umgewandelten Requests zusammen mit noch leeren **HttpServletResponse-Objekten**) entgegennimmt. Der **RequestProcessor** übernimmt dann die Steuerung des Requestablaufs (füllt ActionForm, ruft Action auf, leitet an von Action ausgewählte JSP weiter). Die vollständig implementierten ActionServlet und RequestProcessor bilden den **Front Controller** und steuern die Anwendung anhand von Einträgen in der **Konfigurationsdatei**.
- ◆ **ActionForm:** für die **Zwischenspeicherung** von Formulardaten bzw. elementen (aus Request bzw für Response) sowie einfache **syntaktische Überprüfungen** der Eingabedaten. Anwendung muß abstrakte Klasse ActionForm (als in **Scope** abgelegte **JavaBean**) implementieren.
- ◆ **Action:** Die Klasse Action muss für Anwendungen spezialisiert werden. Führt nach Aufruf durch Front Controller (und Übergabe von in ActionForm abgelegten Formulardaten) **Aufrufe an Model** durch. Nach Lieferung des Modelergebnisses **Auswahl der JSP** für die nächste anzuzeigende Seite (die eigentliche Weiterleitung erfolgt durch Front Controller).
- ◆ **Tag Libraries:** Erleichtern Zusammenspiel JSP mit Struts-Komponenten.
- ◆ **Konfigurationsdatei:** hier legt Entwickler das Zusammenspiel von Struts-Komponenten und JSP's fest, bspw welche Action mit welchem ActionForm bei Abarbeitung Request zusammenarbeiten.
- ◆ **Ressource Bundle:** zu **Internationalisierungszwecken** eingesetzt. Besteht aus Textdateien, welche die in der Applikation vorkommenden Texte in unterschiedlichen Sprachen enthalten. Die **LookupDispatchAction** benötigt Ressource Bundle für Zuordnung HTML-Button->Methode. Übereinstimmende Dateinamen (bis auf Länder/Sprachkennung) mit suffix *.properties*. Einträge nach Muster „Key=Property“ -> **JSP's** enthalten dann keine Texte mehr, sondern lediglich **Verweise („Keys“)** auf die entsprechenden **Texte („Properties“)** im Ressource Bundle.

Request-Ablauf

Nutzer hat Formular ausgefüllt und **Request** abgeschickt. Als **Response** schickt JSP ein HTML-Dokument an Browser.

Request -> **Front Controller** bzw ActionServlet [mit Hilfe Konfigurationsdatei] -> **Action-Form** (Datenspeicherung und syntaktische Überprüfung) -> **Front Controller** -> **Action**

(holt Daten aus Action Form) -> **Model** -> **Action** (speichert Ergebnisse in Scope und wählt JSP) -> **Front Controller** -> **JSP** (greift auf Scope zu und übergibt erstelltes HTML-Dokument an Browser)

Kap10: Konfigurationsdatei

Einleitung

XML-Dokument für Konfiguration des Aufbaus und Ablaufs einer Struts-basierten Webapplikation. Implementierung einer Webapplikation mit Struts besteht (abgesehen vom Model) in Codierung von **ActionForms**, **Actions**, und **JSP's**. Die Konfigurationsdatei enthält Information über die **Beziehung** dieser **Komponenten** (-> wie sie bei Abarbeitung Request zusammenarbeiten).

Einträge

<form-beans>: Definition von ActionForms mit *name* | *type*(=Klassenname) als Attribut

<action-mappings>: für jeden Request einen

<action>-Eintrag: Abbildung eines Requests erfolgt über URL

- **Applikationspfad**->Webapplikation
- zur Applikation **relativer Pfad** -> **<action>**-Eintrag in den **<action-mappings>**
- **suffix .do** -> URL spricht Front Controller der Applikation an

Attribute: *path* | *type* | *forward* | *name* | *scope* | *validate* | *input* | ... | **<forward>**-Eintrag

Kap11: ActionForm

Allgemeines

Ein ActionForm ist eine in einem (Session-)Scope abgelegte **JavaBean**, die **temporären Datenspeicher** darstellt:

- ◆ in Request enthaltene Parameterangaben
- ◆ mit Response gesendete Daten zur Füllung von Formulareingabefeldern.

Kann syntaktische Prüfungen auf den Formulardaten durchführen. Für jedes Webformular -> ein ActionForm. Anzahl reduzieren durch Zusammenfassung von Attributen oder dynamische Eigenschaften.

ActionForm im Request-Ablauf

Für jedes Formularelement -> 1 **Attribut** (sowie getter/setter) sowie **reset(...)**- und **validate(...)** -Methoden (Attribute initialisieren und syntaktische Überprüfungen).

Request -> Front Controller -> evtl AF erzeugen -> reset() -> AF in Scope speichern -> AF mit Formulardaten füllen -> validate() [-> Action -> JSP] [*bei Fehler* -> Eingabe-JSP]

Methoden

reset(...)- und validate(...)- Methode erhalten als **Parameter**:

- ◆ **ActionMapping mapping**: dem Request zugeordneter <action>-Eintrag der Konfigurationsdatei
- ◆ **HttpServletRequest request**: das Request-Objekt (Request-Scope)

reset() dient v.a. der korrekten Verarbeitung von CheckBoxes in Struts

validate() liefert immer **ActionErrors-Objekt** zurück, in dem alle aufgetretenen Fehler mittels *add(String, ActionMessage)* gespeichert werden, um sie später dem Benutzer präsentieren zu können. Ein leeres ActionErrors-Objekt -> bestandene Prüfung

Implementierungsregeln

(1) Zugriffe auf das Modell

Niemals von ActionForm, ausschliesslich durch Actions! -> bessere Komplexitätsbeherrschung durch Trennung von Verantwortlichkeiten. AF ist reiner Datenbehälter für Formulardaten.

(2) Implizite Typkonvertierung

Da HTTP keine Datentypen kennt, werden alle **Formulardaten** als **Zeichenketten** übertragen. Struts bietet implizite **Typkonvertierung** in **primitive Datentypen** zur komfortablen Weiterverarbeitung der Daten an -> automatische Umwandlung in Typ des zugehörigen Attributs des AF.

Mit Vorsicht zu verwenden! **Attribute** eines **AF**, die vom Benutzer auszufüllenden Formularfeldern entsprechen, als **String** vereinbaren. Wenn Benutzer nur auswählen kann (CheckBoxes, RadioBoxes) **passend typisieren**.

Kap12: Actions

Allgemeines

Übernimmt die Steuerung der mit dem Request verbundenen fachlichen Aufgaben. Ruft dafür das Model auf, legt die Ergebnisse für JSP in einem Scope ab und wählt die nächste JSP aus.

„Normale“ Action

Für genau einen Request verantwortlich. Die Zuordnung erfolgt über den zugehörigen <action>-Eintrag in der Konfigurationsdatei. Spezialisierung der Klasse Action mit wichtigster Methode **execute(...)**. Parameter:

- ◆ **ActionMapping mapping**: der dem Request zugeordnete <action>-Eintrag der KonfigD.
- ◆ **ActionForm form**: das zugehörige AF
- ◆ **HttpServletRequest request**
- ◆ **HttpServletResponse response**

Als **return-Parameter** wird ein **ActionForward-Objekt** erwartet, das einen Verweis auf

die nächste (vom FrontController) aufzurufende Action oder JSP beinhaltet.

In KonfigDatei pro AF nur ein <action>-Eintrag. Werden für Request 2 AF benötigt, braucht man 2 Actions, wobei die erste an die zweite weiterleitet (über den Front Controller).

Implementierung der **execute(...)-Methode**

- (1) überprüfen, ob **Benutzer** für Request **authorisiert** ist (zB anhand Session-Information)
- (2) **Benutzerangaben (semantisch) überprüfen** [nach der syntaktischen Prüfung durch AF]
- (3) **Steuerung der eigentlichen fachlichen Aufgaben** in Form von Model-Aufrufen implementieren (Berechnungen, Zustandsabfragen, Datensicherungen...).
- (4) **Ergebnisse** der Modelaufrufe sowie Fehlermeldungen in passendem **Scope** bzw AF **speichern**
- (5) in Abhängigkeit von den Ergebnissen die **nachfolgende JSP** (evtl Action) **auswählen**

DispatchAction

Java-Swing-Framework erledigt die Zuordnung eines Button-Events zur verarbeitenden Methode (das „dispatchen“) transparent für den Entwickler. Auch bei Webapplikationen muss man wissen, welcher Link bzw Button den Event ausgelöst hat.

Wird „normale“ Action eingesetzt, gibt es 2 Möglichkeiten zur Identifizierung eines **Links** -> jeder Link eigene Action | eine Action für alle Links (Identifizierung über Abfragen der Parameter oder Reflection-API).

Die **Dispatch-Action** nimmt dem Entwickler die Arbeit der Identifizierung von **Links** ab: pro HTML -Dokument eine DispatchAction, die für jeden vorkommenden Link eine eigene Methode enthält (mit Signatur wie execute(...)-Methode).

LookupDispatchAction

Wird „normale“ Action eingesetzt erfolgt Identifizierung des auslösenden Buttons über if-then-else-Konstrukte oder Reflection-API, da alle Elemente eines HTML-Formulares stets an ein und dieselbe URL und damit dieselbe Action gesendet werden (im Gegensatz zu Links, die an verschiedene URL's gesendet werden können und somit eigene Actions haben können).

LookupDispatchAction (LDA) sind in Struts speziell für **Buttons** vorgesehen. Für jeden in HTML-Formular vorkommenden Button gibt es in der LDA eine eigene Methode (mit Signatur wie execute(...)-Methode). Die **Auswahl** der zu gedrückten Button gehörenden **Methode** wird vom **Struts-Framework** vorgenommen. Damit das Dispatchen transparent erfolgen kann, muss **Entwickler** dem **Framework** die **Zuordnung Buttons <-> Methoden** über Angaben im **Ressource Bundle** (Hinterlegen der Button-Aufschrift), in **LDA-Klasse** (über HashMap und Methode getKeyMethodMap()) und in der **KonfigDatei** (zusätzliches Attribut parameter in zur LDA gehören <action>) **bekanntmachen**.

ForwardAction

Auf Webseiten finden sich häufig Links mit dem einzigen Zweck auf eine andere Seite zu gelangen. Ein Link von JSP auf JSP würde allerdings den Front Controller umgehen und damit die **Systematik des normalen Request-Ablaufs** durchbrechen. Ein **Link** sollte stets auf eine **Action** verweisen, die eine **JSP** auswählt, an die schliesslich der **Front Controller** weiterleitet. Um die Programmierung quasi leerer Actions zu vermeiden, sieht **Struts** die fertig implementierte **ForwardAction** vor -> ein Eintrag in **KonfigDatei** genügt zur Implementierung.

Implementierungsregeln

- ◆ **Thread-Sicherheit:** Von konkreter Action-Klasse erzeugt Struts-Framework für Dauer einer Applikation immer nur eine einzige Instanz. Mehrere Benutzer teilen sich diese Instanz. Eine **Action** muss wie auch ein **Servlet thread-safe** sein -> 3 Möglichkeiten (s.o.)
- ◆ **Ausnahmebehandlung:** sinnvolle Fehlerbehandlung -> Fehler beheben|verständliche Fehlerseite. Fehlerseiten werden in **KonfigDatei** angegeben.

Kap13: HTML Tag Library

Tag Libraries

Zur Generierung dynamischen Inhalts stellt Struts **5 Tag Libraries** zur Verfügung:

- ◆ **HTML (- Taglib):** Formularerstellung
- ◆ **Bean:** Zugriff auf Ressource Bundle und Erstellen von JavaBeans
- ◆ **Logic:** typische Kontrollflusskonstrukte
- ◆ **Nested:** Kontrollfluss für geschachtelte Objekte
- ◆ **Tiles:** Zusammenarbeit mit Struts Komponente *Tiles* für modulare Realisierung von Strukturierten Seitenlayout (Wiederverwendbarkeit).

Ein Großteil der **Funktionalität** kann inzwischen besser über **EL** und **JSTL** realisiert werden, wodurch die **JSP's** von **Struts unabhängig** werden. Wie alle Tags einer Tag-Library werden **Struts-Tags** zur Laufzeit in die entsprechenden **HTML-Tags** umgewandelt, damit die JSP reine HTML-Dokumente erzeugen kann. Benutzte Taglibs müssen der JSP über **taglib-Direktive** bekannt gemacht werden.

HTML-Taglib

```
<html:form action="..." />
```

definiert Webformular mit Verweis auf <action>-Eintrag

```
<html:hidden property="..." value="..." .../>
```

generiert verborgenes Feld

```
<html:text property="..." />
```

erzeugt Texteingabefeld

```
<html:password property="..." redisplay="..." />
```

erzeugt ein Passwort-Eingabefeld

```
<html:submit property="..." value="..." />
```

generiert (das Formular absendenden) Button

```
<html:cancel value="..." />
```

generiert „Abbrechen“-Button

```
<html:messages id="..." property="..." />
```

zum Anzeigen von Meldungen der *validate()*-Methode des AF oder der *execute()*-Methode der Action

```
<html:link action="..." />
```

erzeugt einen Link

sowie weitere **Tags** für Checkboxes, RadioButtons, Dateiuploads etc. sowie weitere **Attribute** wie *disabled* und *readonly*.

Bean-Taglib

```
<bean:message key="..." ... />
```

Anzeigen der im Ressource-Bundle abgelegten Texte

KE 5

Teil V: Anwendungskern der Java Plattform (JEE 5)

Kap15: Einstieg

Überblick

Nach der **Webschicht** der 5-Schichten-Architektur für Webapplikationen (bzw View und Controller der Model-2-Architektur) werden nun Technologien zur Umsetzung des **Anwendungskerns** (Anwendungslogik und Anwendungsobjekte) vorgestellt.

- ◆ **Anwendungslogik** wird mit **Enterprise JavaBeans (EJB)** realisiert. Diese werden gegenüber reinen Java-(Kontroll)Klassen in **Java-EE-Containern** ausgeführt, von denen sie eine Vielzahl von Dienstleistungen in Anspruch nehmen können (Transaktions- und Sicherheitsmanagement, Lastausgleich in Netzwerken).
- ◆ **(Persistente) Anwendungsobjekte** werden als **Entities** realisiert - vergleichbar mit Instanzen von Entitätsklassen (SE I). Ihre Persistierung erfolgt über die **Java Persistence API**, eine vom **EJB-Container** implementierte **Schnittstelle**, die vom konkreten **Persistenzframework** (zB Hibernate: bildet Klassen auf DB-Tabellen ab und ermöglicht transparentes laden und speichern) und von der **DB** abstrahiert.



Annotationen für Metadaten

Vom EJB-Container benötigte **Meta-Informationen** können seit EJB 3.0 per **Annotationen** direkt im Sourcecode angegeben werden. Die Annotationen werden vom **Java Compiler** zusammen mit dem Quellcode verarbeitet und die Meta-Informationen mit in die compilierten Klassen eingebunden. Haben wie Kommentare keine direkten Auswirkungen auf den annotierten Code, können aber zur **Laufzeit** von anderen Codeteilen (zB einem Framework oder einem Container) durch **Introspektion** ausgelesen werden und somit indirekt doch Einfluss auf das Verhalten einer Software haben.

Eine Annotation wird als **Modifier** vor **Klassen**, **Methoden** oder **Attributen** notiert. Sie ist typisiert: ein **Annotationsbezeichner** wie *Override* bezeichnet einen **Annotationstyp**, der festlegt, in welchem **Kontext** die Annotation benutzt werden darf (vor Klassen Methoden...), welche **Parameter** (Name und Typ) angegeben werden können oder müssen, und wie deren **Default-Werte** lauten. Auf dieser Basis nimmt Compiler **Gültigkeitsprüfung** vor.

- ◆ *@Override*
public boolean methode1 (...)
- ◆ *@Entity*
public class Adresse {...}
- ◆ *@Id @GeneratedValue(strategy=GenerationType.TABLE)*
public long getId() {...}

@Override hat scheinbar keine Auswirkungen, ist aber sehr nützlich (zB bei Schreibfehlern). *@Id* und *@GeneratedValue()* liefern dem Persistenzframework wichtige Informationen.

Annotationen (zB *@EJB*) dienen auch zur **Deklaration der Abhängigkeit (Dependency)** von Komponenten oder Ressourcen => **Dependency Injektion** (angeforderte Injektion zur Laufzeit durch den Container):

```
public class MeinServlet extends HttpServlet {
    @EJB private MaineEJB bean;
    ...
}
```

Kap16: Enterprise JavaBeans (EJB)

Einführung

EJB ist leistungsfähige **Technologie** zur **Realisierung der Anwendungslogik**. Eine EJB ist eine spezielle Java-Klasse, deren Methoden Anwendungslogik implementieren und der Applikation zur Verfügung stellen. Seit EJB 3.0 ist **Implementierung** kaum schwieriger als die von simplen Java-Klassen, bietet aber folgende **Möglichkeiten** (Dienste des EJB-Containers):

- ◆ Wahlweise synchroner oder asynchroner Zugriff

Session Beans bietet über Business Interface **synchron** aufgerufene Business-Methoden an (Thread wartet bis Terminierung Methode und ggf Lieferung eines Rückgabewertes).

Message Driven Bean besitzt kein Business Interface und wird statt dessen bei Nachrichtenwarteschlange angemeldet. Über Nachrichtendienst des Application Servers kann Client so Nachrichten an Message-Driven-Bean senden, welche darauf (**asynchron**) mit der Ausführung von Anwendungslogik reagiert (der Aufrufer arbeitet parallel weiter - ohne direkte Rückmeldung von der Bean).

- ◆ clientspezifischer Zustand

Stateless Session Bean: wird lediglich zur Ausführung einer Business-Methode einem Client zugeordnet. Nach Abarbeitung Methode wird diese Zuordnung wieder gelöst. Die Instanz verbleibt in einem Pool bis sie ein Client wieder benötigt oder bis sie zerstört wird. Sollte keinen Zustand haben -> nur Methoden, aber keine Attribute!

Stateful Session Bean: immer exklusiv einem bestimmten Client zugeordnet. Für jeden neuen Client wird neue Instanz erzeugt. Falls Bean-Klasse Attribute besitzt (ihre Instanzen also einen Zustand speichern), ist sichergestellt das für jeden Client ein spezifischer Zustand (**Conversational State**) nachgehalten wird.

Beans ohne Zustand -> stets stateless: Ressourceneinsparungen und Performanzverbesserungen durch Wiederverwendung der Instanzen (**Pooling**).

Message-Driven-Beans: sind stets stateless.

- ◆ Verteilung mit transparentem Remotezugriff und Lastenausgleich

EJB können auf mehrere Server verteilt werden (Ausfallsicherheit durch Redundanz, Lastenausgleich). Systeme sind skalierbar (Möglichkeit nachträglicher Verteilung).

Message-Driven-Beans eignen sich grundsätzlich zur Verteilung. Bei **Session Beans** wird zwischen **Remote-Zugriff** und **lokalem Zugriff** unterschieden -> zwei Typen von **Business-Interfaces (local und remote)**. Werden Bean und Client in derselben VM ausgeführt: lokaler Aufruf effizienter. Bei Remote-IF: ausschließlich serialisierbare Typen als Parameter verwenden (Übergabe als Kopie „pass-by-value“).

- ◆ Konfigurierbares automatisches Transaktionsmanagement

Entwickler muss keine Transaktionen manuell starten und beenden. **Container** startet transparent eine Transaktion, sobald EJB-Methode von Client aufgerufen wurde und beendet diese beim Beenden der Methode. Dieses **implizite Transaktionsmanagement** ist konfigurierbar (Annotationen oder Deployment-Deskriptoren). Auch **explizites Transaktionsmanagement** durch Programmierer ist möglich.

◆ Sicherheitsmanagement

Kann analog zum Transaktionsmanagement weitgehend dem **Container** überlassen und lediglich **deklarativ** über Annotationen oder Deployment-Deskriptoren **konfiguriert** werden. Bsp: Zuordnung *Benutzerrollen* <-> *Zugriffsrechte*.

Rollenverteilung der Java EE-Plattform:

<i>Rolle</i>	<i>Aufgaben</i>
Component (Bean) Provider	Erstellung Java EE Applikationskomponenten
Application Assembler	Zusammenstellung einer Enterprise Application aus einzelnen Applikationskomponenten
Deployer	Installation von Webapplikationen und EJB's in Container, Konfiguration
System Administrator	Einrichtung, Betrieb, Überwachung des Systems

Annotationen vs. Deployment-Deskriptoren

Erstere sind einfacher und komfortabler, letztere werden aus **Abwärtskompatibilitätsgründen** weiter unterstützt, haben aber auch **Vorteile**: Änderungen sind extern möglich, ohne das der Sourcecode neu kompiliert werden muss. Der **Deployment-Deskriptor** „überstimmt“ **Annotationen** -> ermöglicht Application Assembler bzw Deployer die Modifikation der Voreinstellungen des Komponenten-Entwicklers.

Implementierung einer Session Bean

Beginnt mit Festlegen eines **Business Interfaces**: ein mit javax.ejb.(Local | Remote) annotiertes gewöhnliches Java-Interface. **Einschränkungen**:

- ◆ **Methodenbezeichner** dürfen nicht mit „ejb“ beginnen
- ◆ **Business Methoden** dürfen nicht statisch und nicht final sein, müssen public sein und serialisierbare Parameter haben.

Danach kann die eigentliche **Bean-Klasse** erstellt werden: eine gewöhnliche Java-Klasse, die das **Business-Interface** implementiert (wahlweise @stateful oder @stateless). Sie leitet nicht von einer (abstrakten) Frameworkklasse ab (Framework erkennt sie an Annotation, nicht am Typ), sondern ist (abgesehen von der Annotation) ein **POJO**, kann also insbesondere zum **Aufbau von Klassenhierarchien** benutzt werden. Benötigt **EJB** ein **Objekt vom Framework**, deklariert sie diesen Bedarf durch eine Annotation und lässt sich eine Referenz auf das benötigte Objekt vom Container in eine Variable injizieren.

EJB können neben **Business-Methoden** auch

- ◆ **Lifecycle Callback Methoden**: Ereignisbehandlung (zB vor Zerstörung Beaninstanz)
- ◆ **Interceptor-Methoden**: vor jeder Ausführung einer Businessmethode
- ◆ **Remove-Methode**: Anfordern der sofortigen Zerstörung einer Stateful Session Bean implementieren.

Verwendung einer Session Bean

Zum Zugriff auf die Anwendungslogik benötigt der **Client** (zB der Controller der Webschicht) eine **Referenz** auf eine **Instanz der Bean** (bzw auf ein **lokales Stub-Objekt**, das ebenfalls das Business-IF implementiert), die er vom **Container** anfordert (und nicht mittels *new* erzeugt):

(1) Dependency Injection

In **Client-Klasse** ein **Attribut** vom **Typ** des **Business-IF** deklarieren und mit **@EJB** annotieren. Der **Container** sorgt zur Laufzeit dafür, dass rechtzeitig vor einem Clientzugriff auf dieses Attribut eine Instanz erzeugt bzw einem Pool entnommen wird und eine **Referenz** darauf in das **Attribut** übertragen wird. Der **Client** muss in einem **Java EE-Container** ausgeführt werden, der seine Komponenten introspeziert ((Servlet EJB|Application Client)-Container). **DI** funktioniert also nicht mit **Struts Actions**, da diese keine Java EE-Komponenten sind und nicht in einem Container ausgeführt werden.

(2) Lookup-Anweisungen im Code

Java EE verwendet serverübergreifend arbeitenden **Namens- und Verzeichnisdienst** zur Referenzierung von verteilten Ressourcen, der über die **JNDI Schnittstelle** angesprochen wird. **Ressourcen** werden unter **JNDI-Namen** (in unterhalb von *InitialContext* hierarchisch organisierten Kontexten) verzeichnet und können über einen **Lookup** lokalisiert werden.

Um zB im Rahmen einer *Struts Action* eine *Session Bean* über **JNDI-Lookup** beziehen zu können, müssen **zwei Voraussetzungen** erfüllt sein:

- Die **Bean** muss in einem der Action **zugänglichen Kontext** verzeichnet sein
- Der **Name**, unter dem sie verzeichnet ist, muss **bekannt** sein

Zu **Remote Interfaces** legt der *Sun Application Server* beim Deployment automatisch einen JNDI-Eintrag an und verwendet den qualifizierten Klassennamen als JNDI-Namen.

Zu **Local Interface** wird nicht automatisch ein JNDI-Eintrag angelegt. Für JNDI-Lookup muss der **Client deklarieren**, welche **EJB-Klasse** er benötigt und unter welchem **Namen** er sie im JNDI-Namensverzeichnis erwartet (per Eintrag im **Deployment Descriptor web.xml** [wie *Umgebungsvariable*: <ejb-local-ref>, <ejb-ref-name>, <ejb-ref-type>, <local>], bei introspezierten Servlets auch über **DI**).

Verzeichnisstruktur für das Deployment

Enterprise Applications bestehen aus einer **EJB-Komponente** und einer **Webapplikationskomponente**, die in getrennten Projekten erstellt werden können. Die **Webschicht** hat die vorgegebene Verzeichnisstruktur und kann zum Deployment in einem **WAR (Web Archive)** gebündelt werden.

Die Vorgabe für die **Verzeichnisstruktur** zum Deployment von **EJB's** ist simpler:

<i>Verzeichnis</i>	<i>Inhalt</i>
/Name	Wurzelverzeichnis
/Name/META-INF/	Metadaten über EJB-JAR-Archiv

EJB's und Webschicht können auch zu **Enterprise Archive (EAR)** zusammengefasst

werden:

<i>Verzeichnis</i>	<i>Inhalt</i>
/Name	Wurzelverzeichnis für EJB-JAR's, WAR's und weitere JAR-Files
/Name/lib/	JAR-Bibliotheken die allen Applikationskomponenten gemeinsam zur Verfügung stehen
/Name/META-INF/	Insb. ein gesonderter Deployment-Descriptor für Enterprise Application (application.xml)

Kap17: Entity-Klassen

Einführung

In der 5-Schichten-Architektur folgt nach der (mit EJB implementierten) Schicht für die **Anwendungslogik** die Schicht mit den (**persistenten**) **Anwendungsobjekten**. Sie sind vergleichbar zu Instanzen von Entitätsklassen (SE I).

- ◆ **Transientes Objekt:** flüchtiges, ausschließlich im Arbeitsspeicher und nicht in der DB gespeichertes Objekt.
- ◆ **Persistentes Objekt:** existiert durch Speicherung seines Zustandes in der DB auch nach Beenden der Software weiter. Besitzt **externe** (-> DB-Eintrag) und **interne** (Objekt im Arbeitsspeicher) **Repräsentation**. Wenn beide Repräsentationen existieren, sind sie **synchron**: Änderungen an der internen Repräsentation wirken sich auch auf die externe Repräsentation aus.
- ◆ **Transiente Kopien eines persistenten Objektes** sind Objekte im Arbeitsspeicher, die zwar denselben Zustand haben aber nicht mit den persistenten Objekten synchronisiert sind.
- ◆ **Persistente Klasse:** Java-Klasse, deren Objekte persistent sein können.
- ◆ **Persistieren eines transientes Objektes:** ein transientes Objekt einer persistenten Klasse zu einem persistenten Objekt machen.

Zur Umsetzung der Persistenz von Anwendungsobjektklassen (AO-Klasse oder auch Entitätsklassen) wird die **Java Persistence API** verwendet. Diese ist seit EJB 3.0 unabhängig von den EJB's (Entity Beans gibt es nicht mehr - die Komponenten heißen jetzt schlicht Entities). Sie soll in Zukunft auch in **Java SE** integriert werden, nicht nur in **EJB-** und **Web-Container**.

Die Java Persistence API ermöglicht es, **Java-Klassen** zu implementieren, deren **Objekte** in einem **relationalen DB-System** **persistent** gehalten werden können. Zum **transienten Speichern und Laden** solcher Klassen bindet der implementierende Container ein **Persistenzsystem** wie zB Hibernate oder Oracle TopLink Essentials als **Persistence Provider** ein. Die vom Persistence Provider benötigten Informationen für Abbildung persistenter Klassen auf DB-Relationen werden als **Annotationen** in den Klassen (oder mit **Deployment Deskriptoren**) hinterlegt.

- ◆ **Entity-Klassen:** mit Java Persistence implementierte persistente AO-Klassen
- ◆ **Entity-Instanzen (=Entities):** ihre Instanzen. **Entity** als plattformspezifische (Java)

Implementierung von **Entität** als Objekt plattformunabhängiger Entwurfsklassen unterscheiden.

Erstellen einfacher Entity-Klassen

Wie bei EJB: einfache **Java-Klasse** erstellen und mit **Annotation** als **Entity-Klasse** auszeichnen. Mindestvoraussetzungen:

- ◆ **Klasse**, **Methoden** und der zu persistierende **Zustand** dürfen **nicht final** deklariert sein
- ◆ Klasse muss einen **parameterlosen** (public oder protected) **Default-Konstruktor** haben. Zusätzliche Konstruktoren sind erlaubt.
- ◆ Sinnvollerweise sollte Klasse das IF **java.io.Serializable** implementieren.
- ◆ Es muss ein **Primärschlüssel** definiert werden: **Objekte** werden im Speicher durch Speicheradresse (**Zeiger**) referenziert, **Datensätze** in der relationalen DB jedoch durch **Schlüsselwert**, der Bestandteil des Datensatzes ist. Die **interne Repräsentation** muss ein **Attribut** oder eine **Property** für den **Schlüsselwert** definieren und mit **@Id** (und evtl **@GeneratedValue** mit Parameter **strategy=Generation.Type.(TABLE|AUTO)**) auszeichnen. **Property-based Access** fördert Kapselung und Wartbarkeit gegenüber **Field-based Access**.

Normalerweise werden **alle Properties/Attribute persistent** gehalten. Mit **@Transient** ist aber eine explizite Kennzeichnung möglich, wenn bspw Sitzungsdaten oder abgeleitete Properties nicht persistiert werden sollen.

Für **persistente Properties** wird die **Abbildung auf Felder** einer **DB-Tabelle** für folgende **Typen** unterstützt:

- ◆ numerische Typen und Booleans
- ◆ Strings, Character-Arrays und Byte-Arrays
- ◆ Zeit- und Datumsangaben
- ◆ Aufzählungstypen
- ◆ serialisierbare Typen (implementieren IF Serializable)
- ◆ Typ einer Entityklasse oder einer Menge anderer Entities (-> Assoziationen zwischen Entity-Klassen)

Zur Persistierung einer Entity-Klasse wird vom **Persistence Provider** eine **DB-Tabelle** angelegt, idR **eine Tabelle pro Klasse** (Ausnahme: Klassenhierarchien, bei denen mittels Annotationen zwischen Abbildungsstrategien gewählt werden kann). Die persistenten **Properties** werden auf die **Spalten einer Tabelle** abgebildet (Sonderfälle -> foreign Keys). Die Abstraktion von der verwendeten DB durch das Persistenz-Framework setzt voraus, das die **DB-Schemata** beim Deployment der Applikation vollautomatisch generiert werden. Manuelle Zuordnungen mit Hilfe von Annotationen sind aber ebenfalls möglich.

Beziehungen zwischen Entity-Klassen

Ein Entitätsmodell enthält idR nicht unabhängig nebeneinander stehende

Entitätsklassen, sondern definiert auch **Beziehungen zwischen den Anwendungsobjekten** (implementiert durch Attribute/Properties vom Typ einer anderen Entity-Klasse). Für die **korrekte Persistierung** solcher Beziehungen benötigt die **Persistence API** oft zusätzliche **Meta-Informationen** (in Form von Annotationen).

- ◆ **Unidirektionale 1:1 und n:1 Assoziationen**

einem oder mehreren Anwendungsobjekten einer Klasse A ist höchstens ein Objekt einer anderen Klasse B zugeordnet, und die Assoziation ist nur in Richtung A->B navigierbar.

Implementierung: Klasse A erhält Attribut/Property der Klasse B. Durch Annotationen *@OneToOne* bzw. *@ManyToOne* wird gekennzeichnet, dass Attribute des **Typs B** nicht Inhalt des **A-Objektes** sind (keine Felder der zu A gehörenden DB-Tabelle), sondern **verbundene Entities** mit eigenen DB-Tabellen, die im Datensatz eines A-Objektes lediglich mittels **Schlüsselwerte** referenziert werden.

- ◆ **Unidirektionale 1:n und m:n Assoziationen**

mengenwertige Assoziationen in unidirektionaler Ausprägung, mittels der Annotationen *@OneToMany* und *@ManyToMany* realisiert. **Klasse A** muss jeweils eine **Menge von B-Objekten** in einer Property speichern, deren Typ **Set** ist, also eine duplikatfreie Menge ohne Ordnung. Auch **Collection**, **List**, **Map** sind möglich. Beim **Listentyp** kann (und sollte) durch *@OrderBy* eine **Sortierung** definiert werden, sonst kann beim Auslesen aus DB die Reihenfolge durcheinandergeraten.

- ◆ **Bidirektionale Assoziationen**

Alle unidirektionalen Assoziationen können auf **bidirektionale Navigierbarkeit** erweitert werden, indem auch der **Klasse B** ein geeignet annotiertes **Attribut vom Typ A** hinzugefügt wird. Die **Java Persistence API** bietet einen **Mechanismus**, eine der beiden **gegenläufigen Referenzen** als **Umkehrung** der anderen zu **deklarieren** und so dem Persistenzsystem mitzuteilen, das beide zusammen eine bidirektionale Assoziation realisieren. Bei **1:n** und **n:1** ist immer die **n-Seite** die „**Owning Side**“ und die **1-Seite** die „**Inverse Side**“ (deren Assoziations-Annotation der String-Parameter *mappedBy* hinzugefügt wird). Bei **1:1** und **m:n** können beide Seiten „**Owning Side**“ sein. Die auf **DB-Ebene** erzeugte logische bidirektionale Assoziation wird auf **Objektebene** allerdings durch 2 gegenläufige unidirektionale Referenzen realisiert. Die **Anwendungslogik** ist verantwortlich für die **Konsistenz** der gegenläufigen Referenzen.

- ◆ **Lazy Loading**

Im Normalfall ist für alle Assoziationen (auch mengenwertige) **Eager Loading** vor eingestellt -> zu jeder Entity aus DB werden alle verbundenen Entities ebenfalls geladen, so dass das gesamte Objektgeflecht im Speicher liegt. Beim **Lazy Loading** werden verbundene Entities erst bei Bedarf (bei Zugriff auf Verbindung) dynamisch nachgeladen. **Alle Assoziationstypen** kennen den Parameter *fetch=FetchType*. (*LAZY|EAGER*). Dem verringerten Aufwand stehen beim Lazy Loading **Nachteile** gegenüber: dynamisches Nachladen ist nur innerhalb derselben Transaktion möglich, danach ist Ergebnis nicht mehr definiert.

- ◆ **Kaskaden**

Das automatische Durchreichen von Operationen von einer Entity an verbundene Entities, zB beim **Löschen** im Fall von Existenzabhängigkeiten (zB Komposition) oder beim **Persistieren**. Den Assoziations-Annotationen wird ein **Parameter** hinzugefügt (zB `cascade=CascadeType.REMOVE`).

- ◆ **Klassenhierarchien**

Neben **Assoziationsbeziehungen** gibt es zwischen Klassen auch **Generalisierungsbeziehungen**. Die **Java Persistence API** unterstützt die Persistierung von **Klassenhierarchien** aus **Entity-Klassen**. Lediglich die **Wurzelklasse** definiert einen **Primärschlüssel** mit `@Id`, alle **Subklassen** erben diesen Schlüssel. **Abstrakte Entity-Klassen** sind möglich. Entity-Klassen können Subklassen einer **transienten Klasse** sein, der geerbte Zustand bleibt dabei transient.

Lebenszyklus einer Entity

- ◆ Um eine **neue persistente Entity** zu erstellen, ist zunächst mittels Konstruktor eine **transiente Instanz** der Entity-Klasse zu erzeugen (**Zustand *new***). Dann (innerhalb Transaktion) mittels **Operation *persist*** persistieren. Solange Transaktion noch nicht abgeschlossen: **Zustand *managed***. Bei **commit** -> externe Repräsentation in DB festgeschrieben, das Objekt im Hauptspeicher wird jedoch von persistenter Entity entkoppelt (**detached** -> keine interne Repräsentation mehr, sondern nur noch transiente Kopie der Entity). Bei **rollback** -> Entity nicht persistiert, bleibt transient.
- ◆ **Nach Applikationsstart** innerhalb einer **Transaktion** eine **interne Repräsentation** einer persistenten Entity **beziehen**: im ***managed-Zustand***, solange Transaktion andauert (Entity kann modifiziert werden, dynamisches Nachladen für Lazy Loading ist möglich). Änderungen durch **Commit** bestätigt oder durch **Rollback** verworfen. Nach **Abschluß Transaktion** ist Objekt im Speicher wieder detached. Nach **Commit** ist entkoppeltes Objekt eine transiente Kopie der persistenten Entity, nach **Rollback** enthält es dagegen immer noch die verworfenen Änderungen. Rollback verhindert persistierung in DB, macht Änderungen am Java-Objekt aber nicht rückgängig. Auf entkoppelten Objekt kein Lazy Loading, lediglich Zugriff auf available state.
- ◆ **Entfernen einer persistenten Entity**: im ***managed-Zustand*** durch **Operation *remove*** -> merkt die externe Repräsentation zum Löschen vor (Hauptspeicher-Objekt wird nicht entfernt) => ***removed-Zustand***. Beim **Commit**: externe Repräsentation gelöscht, aus interner Repräsentation wird transientes Objekt im detached-Zustand (durch Garbage Collection gelöscht).

Operationen wie *persist* und *remove* werden vom **Entity-Manager** zur Verfügung gestellt, der eine als **Persistence Context** bezeichnete Menge von persistenten Entities verwaltet, die nicht dupliziert sind und über ihren Primärschlüssel identifizierbar. Zu jeder Entity existiert höchstens 1 Objekt der Klasse im Zustand ***managed*** (aber beliebig viele transiente Kopien im Zustand ***detached***). Die voreingestellte Lebensdauer ist der **transaction-scoped Persistence Context**, ein **extended Persistence Context** ermöglicht die Verwaltung eines Persistence Context über Transaktionsgrenzen hinweg.

Eine **Persistence Unit** ist eine Menge von Entity-Klassen, die eine zusammenhängende,

nicht aufteilbare Einheit bilden (zB durch Assoziationen). Alle Entities müssen gemeinsam in der selben DB abgelegt werden.

Transaktionsmanagement

Viele Operationen zum Verwalten von Entities werden über Entity Manager ausgeführt - wozu **Anwendungslogik** zunächst mal **Zugriff** auf einen solchen **Entity Manager** haben muss. In EJB's lassen sich **container managed Entity Manager** über DI beziehen, welche automatisch durch den Container bereitgestellt werden und implizit das Transaktionsmanagement des EJB-Containers über die **Java Transaction API (JTA)** verwenden. Alternativ können **application managed Entity Manager** verwendet werden.

Bei Container-Transaktionsmanagement startet Container eine Transaktion, sobald Anwendungslogik vom Controller gestartet wird, und beendet diese nach Abschluss mit Commit oder Rollback.

Ausserhalb von **Transaktionen** und damit außerhalb von **Anwendungslogik** können keine Entities im **managed-Zustand** existieren. Aus der höher gelegenen **Webschicht** wird höchstens auf **transiente Entities** zugegriffen.

Rollback einer Transaktion

Um Commit zu verbieten und Rollback anzufordern muss **Anwendungslogik** die Methode `setRollBackOnly()` des **SessionContext-Interfaces** aufrufen. Die benötigte **SessionContext-Ressource** kann über **DI** bezogen werden:

```
@Resource SessionContext ctx;  
if (...) // Fehlerfall  
    ctx.setRollBackOnly();
```

Eingriffe ins Transaktionsmanagement

Auch das (meistens ausreichende) **implizite Transaktionsmanagement** bietet einige **Eingriffsmöglichkeiten**. Für jede Business-Methode existiert ein **Transaction Attribute**, das die Transactionssteuerung beeinflusst (zB `RequiresNew` -> Methode muss in eigener Transaktion ausgeführt werden):

```
@TransactionAttribute(TransactionAttributeType.REQUIRESNEW)
```

Eine Business-Methode ist die kleinste Einheit, die innerhalb einer Transaktion ausführbar ist (bei **container managed Transaction Demarcation**).

Verwendung von Entities

EJB-Klasse muss zunächst mittels DI einen **Entity-Manager** zur Verwaltung eines **Persistence Context** beziehen:

```
@PersistenceContext EntityManager em;
```

Persistieren einer neuen transienten Entity:

```
try {
```

```
em.persist(<<entity>>)  
} catch (EntityExistsException e) {...}
```

Persistieren von Objektgeflechten

Falls zu persistierende Entity Verbindungen zu anderen Entities eingeht, muss sehr gründlich vorgegangen werden! Eine **neue transiente Entity** kann nur in folgenden Fällen **erfolgreich persistiert** werden:

- ◆ Entity geht noch **keine Verbindungen** ein
- ◆ Sie geht **nur Verbindungen** mit bereits persistierten Entities ein
- ◆ Sie geht zwar **Verbindungen** zu weiteren **neuen transienten Entities** (Zustand *new*) ein, diese werden aber **alle in derselben Transaktion mit persistiert**.

Laden einer persistenten Entity

- ◆ einzelne Entity über **ID referenzieren** und einzeln in den Speicher laden -> mittels *find(Klasse, ID)*-Operation des Entity-Managers.
- ◆ mit **Anfragesprache** (ggf auch leere|einelementige) Menge von Entities zu Suchkriterien ermitteln.

Ändern einer persistenten Entity

Eine **persistente Entity** kann direkt **modifiziert** werden, indem ihre **interne Repräsentation** (Objekt der Entity-Klasse im managed-Zustand) **modifiziert** wird. Auch das Herstellen von Verbindungen zwischen persistenten Entities ist so möglich. Änderungen werden mit dem **Commit** der **laufenden Transaktion** (dem Beenden der laufenden Anwendungslogik-Methode) **automatisch persistiert** -> keine Operation des Entity-Managers benötigt.

Oft liegt Anwendungslogik-Methode jedoch gar keine persistente Entity vor.

Typische Situation: Während **Request** wird Entity geladen und (als **transiente Kopie**) an **Webschicht** zur Präsentation weitergereicht. Der **User** gibt in späterem Request **Änderungen** an Entity in Auftrag, die noch vom **Controller** der Webschicht aus Requestparametern (oder AF) in die **in transiente Kopie übertragen** werden. Die **modifizierte transiente Entity** wird dann vom **Controller** an eine **Businessmethode** übergeben, um Änderungen zu persistieren. Diese könnte über *em.find(...)* anhand der ID der transienten Kopie die **persistente Entity** beziehen und dann die **Änderungen übertragen**, indem jede persistente Property einzeln übertragen wird.

Mit *em.merge(<<transiente Entity im detached-Zustand mit ID>>)* wird das Einpflegen des Zustandes einer transienten Entity in die DB deutlich komfortabler. Die Operation *merge()* automatisiert den oben beschriebenen Vorgang.

Sind nicht alle verbundenen Entities einer Instanz (im detached-Zustand) einer Klasse mit **Lazy Loading** geladen, muss **Persistence Provider** beim **Merge** sicherstellen, dass die betroffenen Verbindungen unverändert bleiben.

Wird *merge()* eine **transiente Entity** übergeben, zu der **keine** zu aktualisierende **persistente Entity** existiert, legt die Operation eine **neue persistente Kopie** an -> die übergebene **transiente Kopie** behält ihren (*detached*-)Zustand und nimmt -anders als bei

persist()- nicht den *managed*-Zustand an.

Löschen einer persistenten Entity

Das Entfernen einer persistenten Entity aus einem Persistence Context und somit die Zerstörung ihrer externen Repräsentation geschieht mit Hilfe der *remove*-Operation des zugehörigen Entity-Managers, dem die interne Repräsentation im *managed*-Zustand der zu löschenden Entity zu übergeben ist. Wie beim Ändern muss der **Business-Methode**, die löschen soll, die ID der Entity übergeben werden, welche dann über *em.find* (<<Klasse, ID>>) bezogen wird.

Remove() auf Entity im *detached*-Zustand löst Exception aus, auf Entity im *new*-Zustand bleibt wirkungslos (außer bei Kaskaden).

Eine persistente Entity kann nur gelöscht werden, wenn sie entweder gar keine Verbindungen eingeht oder von ihr lediglich unidirektionale Verbindungen zu anderen persistenten Entities ausgehen. Wird sie jedoch von anderen persistenten Entities referenziert, würden diese Referenzen ungültig, das DB-System würde den inkonsistenten Zustand beim Commit erkennen **RollbackException** auslösen. Somit sind vor dem Löschen bidirektionale oder ankommende unidirektionale Assoziationen mit anderen Entity-Klassen -evtl auch die verbundenen Entities- zu entfernen. Remove-Kaskaden bei Kompositionen bilden einen Sonderfall -> Verbindungen brauchen nicht gelöst zu werden.

Prüfen, ob Entity persistent/ managed ist

Für das Ändern und Löschen persistenter Entities ist es wesentlich, dass das dem Entity-Manager übergebene Objekt im Zustand *managed* (nicht *detached*) ist. Die Operation *em.contains(...)* überprüft zur Laufzeit, ob Entity sich im Persistence Context befindet.

Anfragen (Queries) über Entities

em.find(...) hat 2 große Nachteile:

- ◆ ID der Entity muss bekannt sein
- ◆ es kann immer nur genau eine Entity geladen werden, keine Mengen von Entities.

Die Java Persistence API bietet eine eigene Anfragesprache, die **Java Persistence Query Language**, in der man Anfragen auf Entity-Klassen anstatt auf DB-Tabellen formulieren kann - und somit die Abstraktion von der verwendeten DB erhalten bleibt und eine Abbildung der Ergebnisdatensätze auf Entities wegfällt. Beispiel:

```
Query q = em.createQuery( „SELECT ...“ );  
return q.getResultList();
```

Deployment einer Persistence Unit

Eine Persistence Unit besteht im Wesentlichen aus einer Menge von Entity-Klassen. Sie kann als Teil eines EJB-JAR oder auch als separate JAR gepackt werden. Die Verzeichnisstruktur ist praktisch identisch zu der eines EJB-JARs.

Konfigurationsmaßnahmen zum Deployment

Es genügt leider nicht, EAR-File auf Application-Server zu deployen. Es muß mindestens eine Verbindung zu einem DB-Server eingerichtet werden. Auf Applicationserver sind hierzu zunächst Ressourcen anzulegen:

- ◆ DB-System festlegen
- ◆ Verbindung zum DB-Server konfigurieren
- ◆ Datenbank spezifizieren

In Deployment Deskriptor *persistence.xml* wird **JNDI-Name** der **Datenbank** eingetragen sowie die Verwendung der **Java Transaction API (JTA)** eingestellt. Weiterhin wird eine Anweisung hinterlegt, dass **DB-Schemata** automatisch beim Deployment der Persistence Unit aus deren Entity-Klassen zu **generieren** und die **DB-Tabellen** entsprechend **anzulegen** sind.

Kap18: Ausgewählte Entwurfsmuster

Allgemeines

Ähnlich wie Architekturmuster beschreiben **Entwurfsmuster** auf relativ abstrakter Ebene gängige **Konzepte** zur **Lösung weit verbreiteter Entwurfsprobleme** (Bsp: Front Controller Pattern).

Session Fassade

Schnittstelle zur Anwendungslogik auf Seite des **EJB-Servers**.

Fassade beschreibt eine Klasse, die eine ausgewählte **Teilmenge der Funktionalität** eines komplexen **Subsystems** über eine **zentrale Schnittstelle** anbietet. Details des Subsystems werden hinter der Fassade verborgen.

Eine **Session Fassade** sieht Session Bean als Fassade für einen (fachlich zusammenhängenden) Teil des Java EE-Anwendungskern vor -> Zugriffe eines Clients auf Anwendungslogik/-objekte erfolgt dann ausschliesslich über Fassaden.

=> Reduktion der Kopplung und leichtere Wartung.

Business Delegate

Schnittstelle zur Anwendungslogik auf Seiten des **Clients**. Gehört zur Webschicht (Controller).

Alle **Zugriffe von Controller-Klassen** (zB Struts Actions) auf **Anwendungslogik** nicht mehr als direkte Zugriffe auf EJB-Business-Methoden, sondern als **clientlokale Methodenaufrufe** auf dem **Business-Delegate**. => Reduktion der Kopplung und leichtere Wartung, Performance-Verbesserungen (durch Caching).

Gut mit Session Fassade kombinierbar:

- ◆ **Business Delegate** übernimmt auf Clientseite die Abstraktion von Verteilung und Technologie des Anwendungskerns und delegiert Anwendungslogik-Aufrufe an Session

Fassade weiter

- ◆ **Session Fassade** sorgt auf Serverseite für die Weiterverarbeitung und delegiert an feingranularere Business-Methoden.

Service-Locator

Business Delegate stützt sich idR auf **Service Locator** ab. **Gleichartige Tätigkeiten** in den einzelnen Business-Delegate Klassen (zB Lookup einer Session mit JNDI-InitialContext-Objekt) werden zur Redundanzvermeidung und Performanceverbesserung **zentral** in **separate Klasse** ausgelagert. Service Locator zählt zur Webschicht.

Data Transfer Objects (DTO)

Ein **serialisierbares Objekt** zur Aufnahme einer **Sammlung von Daten**, die gebündelt zwischen 2 Kommunikationspartnern übertragen werden -> reduziert Anzahl Kommunikationsvorgänge (zB zwischen Webschicht und Anwendungslogik, indem DTO vom Anwendungskern zum Controller und von dort zur View übergeben wird). Insbesondere für **Remote-Zugriffe** empfohlen.

ActionForm ist eine **Struts-spezifische Ausprägung des DTO-Musters** -> gebündelte Übertragung von Daten zwischen View (JSP) und Controller (Action).

Value List Handler

Bei Suchen über größere Datenmengen stellt die **Darstellung umfangreicher Ergebnislisten** bei Webapplikationen ein **Problem** dar: **unübersichtliche Webseiten** und **lange Transferzeiten** zum Browser (und bei interner Verteilung auch zwischen Webschicht und Anwendungskern). **Lösung**: Ergebnisliste ausschnittsweise präsentieren und User die Möglichkeit geben, zu weiteren Ausschnitten zu wechseln.

In Anwendungslogikschicht wird eine **Value List Handler-Klasse** realisiert (bspw als stateful session bean -> gepufferte Ergebnisliste ist clientspezifischer Conversational state). Eine **Instanz** dieser Klasse startet auf Client-Anforderung eine **Suche** (durch Delegation), speichert das gesamte Suchergebnis und ermöglicht es ihrem Client über kompakte Suchergebnisse zu iterieren. Die gesamte Ergebnisliste wird lediglich serverintern ermittelt und verwaltet, jedoch nie en bloc über Netzwerkverbindung übertragen (auch nicht intern).

Kombination der Entwurfsmuster

Webschicht:

- ◆ ein **Business Delegate Objekt**, auf dem Controller Daten anfordert.
- ◆ Ein **Service Locator**, der eine Referenz auf die Fassade liefert.

Anwendungskern:

- ◆ korrespondierende **Session Fassade** (Session Bean), an deren gleichnamige Methode der Business Delegate Aufruf delegiert wird (sofern Ergebnis nicht bereits im Cache des Business Delegate Objekt vorliegt).
- ◆ Ein **Data Transfer Object**, in dem die Session Fassade die Ergebnisse von Anwendungskernaufrufen bündelt und an die Webschicht zurück transferiert.

- ◆ Ein **Value List Handler** könnte selbst eine Session Fassade darstellen oder hinter einer solchen verborgen werden. Zur Übertragung von Ergebnis-Teillisten kann DTO herangezogen werden. Eine Business Delegate Klasse mit Cache auf Seiten der Webschicht vermeidet das wiederholte Übertragen von Ergebnis-Teillisten.

KE 6

Teil VI: Interaktionsorientierte UML-Aktivität

Kap19: UML-Aktivität

Motivation

In der **Anforderungsermittlung** erfolgt die **Spezifikation von Anwendungsfällen (use cases)** zunächst als möglichst verständliche **textuelle Beschreibung** (Name, Akteure, VB, NB, normaler|alternativer Ablauf, Ausnahmesituationen).

Als Vorgabe für die Realisierung werden die Anwendungsfälle in der **Analyse** auf der Basis des Analyseklassenmodells mit **Interaktionsdiagrammen (Sequenz-/Kommunikationsdiagrammen)** präzisiert. Dabei geht einerseits die **Realisierungsunabhängigkeit verloren**, andererseits bleibt die **Spezifikation unvollständig**, da pro Interaktionsdiagramm immer nur ein einziges Szenario (ein einziger konkreter Durchlauf durch Anwendungsfall) dargestellt wird.

UML-Aktivitäten eignen sich zur Modellierung von Abläufen **ohne Realisierungsaspekte** - also insbesondere für die Anforderungsermittlung. Ein **Aktivitätsdiagramm** kann auch komplexe Abläufe mit vielen Verzweigungen, Wiederholungen etc. übersichtlich darstellen. **Sämtliche möglichen Abläufe** eines **Anwendungsfalles** können in einem Diagramm veranschaulicht werden.

Grundlagen

Ein **Aktivitätsdiagramm** ist ein **gerichteter Graph**, dessen Knoten durch gerichtete Kanten verbunden sind, die den Kontrollfluss darstellen -> deutliche Ähnlichkeit zu **Kontrollflussgraphen**. Es können globale **VB** und **NB** angegeben werden. Zwischen **Start-** und **Endknoten** entwickeln sich der **Main Flow** sowie **Exceptional Flows** (mit eigenen lokalen NB) mit folgenden **Knotentypen**:

- ◆ **Aktionsknoten (Action)**: elementarer Schritt | Subaktivität (Symbol: Gabel) -> auch durch Aktivitätsdiagramme veranschaulicht, wodurch **Aktivitätshierarchien** entstehen
- ◆ **Zusammenführungsknoten (MergeNode)**: mehrere eingehende, eine ausgehende Kante -> Zusammenführung von Kontrollflüssen.
- ◆ **Entscheidungsknoten (DecisionNode)**: eine eingehende, mehrere (mit disjunkten Be-

dingungen versehene) ausgehende Kanten -> Verzweigen von Kontrollflüssen. Ausserdem gibt es noch **Notizen (Comment)** für beliebigen Text sowie **Konnektoren (connector)** für mehrseitige Diagramme und sonstige darstellungsbedingte Kantenunterbrechungen.

Objekte und Objektfluss

Bei der Modellierung ablauforientierten Verhaltens müssen **Daten** bzw **Objekte** miteinbezogen werden. Aktivitäten bieten hierfür:

- ◆ **Objektfluss:** Kanten, an denen Objekte entlangfließen (die mit Objektknoten verbunden sind)
- ◆ **Objektknoten:** Objekte eines bestimmten Typs (optional in angegebenen Zustand), werden von vorangehenden Aktionen geliefert (die Knoten selbst produzieren/duplizieren keine Objekte) und von nachfolgenden Aktionen konsumiert.
- ◆ **Pins:** spezielle **Objektknoten**, die als kleines Rechteck zwischen Aktion und Objektfluss positioniert werden und mit dem Objekttyp annotiert werden. **Eingabe-Pins** entsprechen Eingabeparametern einer Aktion, **Ausgabe-Pins** den Ausgabeparametern. Es können mehrere Pins an jede Aktivität (auch Subaktivität) geheftet werden. Auch **Entscheidungsknoten** können Aktivität mit Pin haben (max 1 inParameter, genau 1 outParameter). Dazu kommt eine Notiz <<DecisionInput>>.

Parallelität

Mit **Aktivitäten** können auch **parallele Abläufe** (oder sequentielle Abläufe, die vor der nächsten Aktion beendet sein müssen) **modelliert** werden:

- ◆ **Parallelisierungsknoten (ForkNode):** eine eingehende, mehrere ausgehende Kanten
- ◆ **Synchronisierungsknoten (JoinNode):** mehrere eingehende, eine ausgehende Kante.
- ◆ **Implizite Parallelisierung:** gehen mehrere Flüsse (Pins) aus Aktion heraus, startet jeder Fluss einen eigenen parallelen Fluss.
- ◆ **Implizite Synchronisierung:** gehen mehrere Flüsse (Pins) in Aktion herein, wird nachfolgende Aktion erst gestartet, wenn alle Flüsse bzw Objekte vorliegen.

Auch Objektflüsse können parallelisiert werden (dasselbe Objekt wird mehreren Aktivitäten zur Verfügung gestellt).

- ◆ **Ablaufendknoten (FlowFinalNode):** beendet nicht gesamte Aktivität, sondern lediglich alle hineingehenden Kontroll- und Objektflüsse.
- ◆ **Kontrollknoten:** Oberbegriff für Start-, End-, Zusammenführungs-, Entscheidungs-, Synchronisierungs-, Parallelisierungs- und Ablaufendknoten

Kap20: Anpassung an interaktive Systeme

Allgemeines

Interaktionsorientierte Aktivitäten sind eine **Anpassung** der **UML-Aktivitäten** an die Modellierung von Anwendungsfällen im Kontext **interaktiver Systeme** (zB Webapplikationen). Sie bilden die Grundlage für (später eingeführte) auf den Entwicklungsprozess

zugeschnittene Aktivitäten. Die Anpassung erfolgt durch **Stereotypen** und **Szenen**.

Stereotypen

Dienen der Unterscheidung der Aktionen von Akteuren und System.

- ◆ **Akteuraktion:** menschlicher Akteur füllt Eingabefelder auf Bildschirm aus und drückt Buttons oder Links zur Ausführung. Wird als Aktionsknoten mit Stereoty <<AA>> dargestellt.
- ◆ **Systemaktion:** Aktion die von System eigenständig ausgeführt wird. Wird als Aktionsknoten mit Stereoty <<SA>> dargestellt.

Bei **Subaktivitäten (Gabel-Symbol)** differenziert man zwischen

- ◆ eigenständigen Anwendungsfällen (Stereotyp <<AWF>>)
- ◆ komplexen Systemaktionen (Stereotyp <<SA>>)

Szenen

Dienen der **Beschreibung der UI** in Form spezieller **Notizen** mit dem **Schlüsselwort <<Szene>>**, die (rechts) an Aktionen von Akteuren geheftet werden. Dadurch werden für den Ablauf von Anwendungsfällen relevante Aspekte der UI in die Aktivitäten integriert. Die **Informationen** werden in **4 Kategorien** aufgelistet:

- ◆ <<**angezeigte Attribute**>> Attribute, deren Werte vom System dargestellt werden und für Benutzer unveränderlich sind.
- ◆ <<**zu editierende Attribute**>> Attribute, deren Werte vom System dargestellt werden und für Benutzer modifizierbar sind.
- ◆ <<**einzugebende Attribute**>> Attribute, deren Werte allein der Benutzer bereitstellt.
- ◆ <<**(selektierbare | editierbare | einzugebende) Liste**>> Mengen von Attributen, deren Werte zunächst lediglich dargestellt werden.

Teil VII: Überblick über die Methode ABWeb

Kap21: Überblick über die Methode ABWeb

Allgemeines

Bei **ABWeb (Aktivitäts-basierte-Entwicklung von Webapplikationen)** handelt es sich um eine modellbasierte Methode, welche die Softwareerstellung als Folge systematischer Transformationen von Modellen begreift. Neben dem **Domänenklassenmodell** bildet vor allem die **interaktionsorientierte Aktivität** das grundlegende Modell, daher der Name.

Anforderungsermittlung

Für die **integrierte Modellierung** von Anwendungsfällen und Benutzungsschnittstelle

werden **anforderungsorientierte Aktivitäten** verwendet, die gleichzeitig auch das zentrale Modell für die **Validierung** sind. Die Anforderungsermittlung fokussiert auf die **funktionalen Anforderungen**.

Das **Domänenklassenmodell** beschreibt die relevanten Dinge der Problemwelt in Form von Klassen und deren Beziehungen. **Operationen** bleiben **unberücksichtigt** (wg der unerwünschten Realisierungsaspekte), weshalb eine grosse Ähnlichkeit mit dem **Entity-Relationship-Modell** gegeben ist.

Um die Anzahl der Knoten zu verringern wird vom System zwischen zwei Akteurknoten ausgeführte komplexe Anwendungslogik in einem einzigen **komplexen Systemknoten** zusammengefasst, dessen Spezifikation in ein separates Dokument ausgelagert wird (als Text oder als Systemaktivitäten).

Abschliessendes Dokument der **Anforderungsermittlung** bildet die **Anforderungsspezifikation** (Spezifikation aller zu realisierenden funktionalen Anforderungen) mit den zentralen Bestandteilen:

- ◆ anforderungsorientierte Aktivitäten
- ◆ Systemaktivitäten
- ◆ Domänenklassenmodell

Dazu kommen textuelle Beschreibungen der Anwendungsfälle, Anwendungsfallmodelle, Navigationsmöglichkeiten und globales Bildschirmlayout.

Softwarespezifikation

Die **Softwarespezifikation** dient als Ausgangsbasis für eine systematische Transformation in das **Klassenmodell** des **Softwareentwurfs**. Dabei werden die **anforderungsorientierten Aktivitäten (aoA)** in software-spezifikationsorientierte Aktivitäten (**sw-spezifikationsorientierte Aktivitäten ssoA**) überführt, indem im Wesentlichen zusätzliche Systemaktionen eingefügt werden. Die zugunsten besserer Lesbarkeit aufgegebenene **UML-Verträglichkeit** wird dabei wiederhergestellt.

Alle Einzelschritte der Transformation sind rein technischer Natur und können automatisiert werden. **ssoA** erlauben eine besonders einfache Abbildung in das Entwurfsklassenmodell.

Dokumente der Softwarespezifikation: Die **ssoA**, die **Spezifikation der Systemaktivitäten** sowie die unveränderten Dokumente der Anforderungsspezifikation (**Domänenklassenmodell** und die **ergänzenden Modelle und Beschreibungen**)

Softwarearchitektur

ABEweb basiert auf der **5-Schichten-Architektur**. Die drei inneren Schichten der Architektur werden mit dem **MVC-Pattern** überlagert:

- ◆ **Webschicht:** in **MVC-Controller** und **MVC-View** verfeinert
- ◆ **MVC-Model:** gemäß Schichten **Anwendungslogik** und **Anwendungsobjekte** aufgeteilt.

Softwareentwurf

Ausgangspunkt für den Softwareentwurf bilden einerseits die **Softwarespezifikation** und andererseits die **Softwarearchitektur**. Gesteuert von der Softwarespezifikation

verfeinern **Entwurfsentscheidungen** die Architektur, sie verändern diese nicht (sie sind **architekturkonform** und in Bezug auf die Architektur **lokaler Natur**).

Der **Entwurfsansatz von ABWeb** besteht darin, die **ssoA** und die **Systemaktivitäten** sowie das **Domänenklassenmodell** systematisch in ein auf **Struts** und **Java EE** basierendes **Entwurfsklassenmodell** zu transformieren.

Die **Transformation der ssoA** stellt eine **1:1 Abbildung** der Knoten und Kanten der Aktivitäten in Klassen, Operationen und Abhängigkeiten (Dependencies) der Webschicht dar. Eine Szene kann auf bis zu drei Klassen abgebildet werden.

Die Modellierung mündet in eine **Entwurfsspezifikation**, deren zentrales Dokument das **architekturkonforme Entwurfsklassenmodell** ist.

Teil VIII: Anforderungsermittlung

Kap22: Anforderungsermittlung im Allgemeinen und bei ABWeb

Anforderungsermittlung im Allgemeinen

Bei der **Anforderungsermittlung** stehen die Fragen nach dem **WAS** (Funktionen), **WARUM** (Ziele) und **WOMIT** (Objekte und Beziehungen der Problemwelt) im Vordergrund. Das **WIE** (Realisierung) spielt noch keine Rolle. Man unterscheidet **funktionale Anforderungen** (Produktfunktionen und Produktdaten des Systems) und **nicht-funktionale Anforderungen** (Qualität, Performance, Technik).

Der **Auftraggeber** erstellt ein **Lastenheft**, das aber als Vorgabe für die Entwicklung weder präzise noch umfassend genug ist, weswegen eine genauere **Anforderungsermittlung** notwendig ist.

Bei dieser wird zuerst ein **Grundverständnis** der Domäne und der projektspezifischen Anforderungen erarbeitet und ein **Glossar** mit einer Sammlung aller wichtigen materiellen Gegenstände und immateriellen Sachverhalte der Domäne erstellt (**Glossar** ≠ **Datenlexikon**, das weniger umfangreich, aber präziser ist). Das Glossar vereinfacht die Kommunikation der Stakeholder, kann mit den Entwicklungsdokumenten verknüpft werden und spielt eine zentrale Rolle bei der Erstellung der Benutzerdokumentation.

Teilaufgaben der Anforderungsermittlung:

- ◆ **Extraktion** der Anforderungen (Elicitation)
- ◆ **Verhandlung** (Negotiation) über Art und Umfang der Anforderungen
- ◆ **Spezifikation** der Anforderungen in Referenzdokument
- ◆ **Validierung** der Anforderungen auf Vollständigkeit und Korrektheit (sowie erste Verifikationsaktivitäten -> Konsistenz der Teilmodelle)

Je nach **Vorgehensmodell** bezieht sich die Teilaufgabe auf das **Gesamtsystem** (Wasserfallmodell) oder nur die gerade durchgeführte **Iteration** (Rational Unified Process).

Die ermittelten Anforderungen werden in der **Anforderungsspezifikation** zusammengefasst, die als **Vorgabe** für **System-Realisierung** und abschliessenden **Ab-**

nahmetest fungiert.

Alle Tätigkeiten der Softwareentwicklung müssen die Entwicklung der **Benutzungsschnittstelle** systematisch mit einbeziehen, da sowohl **Benutzungsoberfläche** als auch **Benutzerinteraktionen** und **Navigationsmöglichkeiten** für die Akzeptanz einer Anwendung von entscheidender Bedeutung sind. Die Anforderungsermittlung konzentriert sich auf grundsätzliche Eigenschaften inklusive Benutzbarkeitsüberlegungen. Die **Qualität der Anforderungsermittlung** bestimmt in hohem Maße den Erfolg des Gesamtprojektes. Fehler sind hier teuer und können zum Scheitern führen.

Anforderungsermittlung bei ABWeb

ABWeb legt den Schwerpunkt auf die **Modellierung der funktionalen Anforderungen**. Ausgangspunkt ist ein **Lastenheft** mit idR vier Rubriken (Zielbestimmung und Zielgruppe, funktionale Anforderungen, nicht-funktionale Anforderungen, Sonstiges). Nach dem Erstellen eines **Glossars** und (bei großen Systemen) Aufteilung der Funktionalität in **Funktionsbereiche** werden aus **Szenarien Anwendungsfälle** entwickelt und als möglichst verständliche **anforderungsorientierte Aktivitäten (aoA)** modelliert. **Komplexe Systemknoten** werden dabei als **Text** oder **Systemaktivitäten** ausgelagert. Die aoA werden um textuelle Beschreibungen der Anwendungsfälle sowie Anwendungsfallmodelle ergänzt.

Grundlage der **Modellierung der Benutzungsschnittstelle** bilden die **stereotypisierten Knoten** in den Aktivitäten (Akteur- und Systemknoten) und **Szenen**. Zudem werden das **globale Bildschirmlayout** und **Navigationsmöglichkeiten** festgelegt. **Anwendungsfälle** und **UI** sollten möglichst mit integriertem Ansatz modelliert werden. Die daraus hervorgehenden **aoA** werden im Hinblick auf **Benutzbarkeitsgesichtspunkte** analysiert (uU mit Hilfe von **Mockups**).

Die für das **Domänenklassenmodell** relevanten **Elemente** der Domäne (zB in Szenen auftretende Objekte bzw Daten) werden bereits bei der **Modellierung der aoA** notiert. Sie werden mit den im Lastenheft angeführten **Produktdaten** und den **Glossareinträgen** abgeglichen, geeignet ergänzt und dann zur Bildung der Domänenklassen herangezogen. Das **Domänenklassenmodell** hat große Ähnlichkeit mit dem **Entity-Relationship-Modell**, da es wg der unerwünschten Realisierungsaspekte auf **Operationen verzichtet**.

Die resultierende **Anforderungsspezifikation** umfasst folgende Bestandteile:

- ◆ anforderungsorientierte Aktivitäten (aoA)
- ◆ Systemaktivitäten
- ◆ Domänenklassenmodell
- ◆ Ergänzende Modelle und Beschreibungen

Sie soll einen vollständigen Überblick über die Anwendungsfälle und Domänendaten geben, die Stakeholder-Kommunikation erleichtern und die Anforderungvalidierung ermöglichen.

Merkregel: Jede Modellierungsaktivität in der Anforderungsermittlung orientiert sich ausschließlich an der Problemwelt (Problemadäquatheit), darf nicht Selbstzweck werden und sollte keine Realisierungsgesichtspunkte vorwegnehmen. Modellelemente hinterfragen, die kein Pendant in der Problemwelt besitzen.

Kap23: Integrierte Modellierung Anwendungsfälle und UI

Anforderungsorientierte Aktivitäten

aoA integrieren die Modellierung von Anwendungsfällen und Benutzungsschnittstelle. Aus rein didaktischen Gründen wird die aoA als **Transformation** einer **interaktionsorientierten Aktivität** definiert:

- ◆ **Aktionsknoten** und nachfolgende **Entscheidungsknoten** werden zu Aktionsknoten mit mehreren ausgehenden Kontrollflüssen **verschmolzen**. **Entscheidungsknoten** ohne vorhergehenden Aktionsknoten werden durch einen Aktionsknoten mit mehreren ausgehenden Kontrollflüssen ersetzt. Die entstehenden Knoten werden mit <<AE>> für „**Akteurentscheidung**“ und <<SE>> für „**Systementscheidung**“ stereotypisiert. Die Aktionen sind syntaktisch, aber nicht semantisch UML-konform.
- ◆ Die **Anzahl der Systemknoten** wird **reduziert**. Systemaktionen ohne Anwendungslogik werden weggelassen. Jede Folge von Systemknoten ohne Zusammenführungsknoten wird zu einem einzigen **komplexen Systemknoten** zusammengefasst. Eine Subaktivität, die einen komplexen Systemknoten repräsentiert, wird **Systemaktivität** bzw **Systementscheidung** (unter den zusammengefaßten Knoten gab es mindestens 1 Entscheidungsknoten) genannt.

Eine derart aufbereitete interaktionsorientierte Aktivität wird mit <<**anforderungsorientiert**>> stereotypisiert.

Vorgehen bei Erstellung von anforderungsorientierten Aktivitäten

Ausgangspunkt der Modellierung bilden die im **Lastenhaft** aufgeführten **Produktfunktionen**. Die Stakeholder sprechen gemeinsam **Szenarien** durch und synthetisieren Schritt für Schritt **Anwendungsfälle**, die in **aoA-Diagrammen** festgehalten werden. Hinzu kommen **strukturierte textuelle Anwendungsfallbeschreibungen** und **Anwendungsfallmodelle**. Die anfänglich grobgranularen Diagramme mit vager Semantik werden schrittweise verfeinert und bei Bedarf umstrukturiert. **Subaktivitäten** werden auch unter dem Gesichtspunkt der Wiederverwendbarkeit gebildet. Es empfiehlt sich, parallel zur Festlegung der **Akteurknoten** die zugehörigen **Szenen** zu betrachten (Welche Attribute|Buttons|Links? Wer stellt Attributwerte bereit? Sind Attribute mit Domänenklassenmodell abgeglichen?).

Während der Modellierung werden fortlaufend die **VB** und **NB** skizziert. Für den **normalen** (und die **alternativen**) **Kontrollfluß**(-flüsse) werden VB und NB mit den Schlüsselwörtern <<**precondition**>> und <<**postcondition**>> oberhalb Diagramm notiert, ein **Ausnahmekontrollfluß** wird mit Notizelement am Endknoten gekennzeichnet.

Graphische Stereotypen und die Verwendung **domänenspezifischer Sprachen (DSL)** erhöhen die Benutzerfreundlichkeit.

Systemaktivitäten

aoA sollen möglichst **verständlich** für den Anwender sein. **Komplexe Systemknoten** werden daher in aoA **nicht** näher **präzisiert**. Für die Umsetzung durch die Entwickler müssen sie natürlich ausspezifiziert werden. Ihre **Anwendungslogik** wird zwar **ermittelt**, aber nicht in den aoA sondern in einem **separaten Dokument** als Text oder eigene Ak-

tivität festgehalten. Man unterscheidet zwei Fälle: der komplexe Systemknoten ist eine **Systemaktion** oder eine **Systementscheidung**.

Kap24: Ergänzende Modellierung von Anwendungsfällen und

Benutzungsschnittstelle

Allgemeines

Die aoA in ABEweb integrieren die Modellierung von Anwendungsfällen und Benutzungsschnittstelle. Daneben gibt es aber auch Aspekte, die entweder ausschließlich die Anwendungsfälle oder ausschließlich die UI betreffen.

Textuelle Anwendungsfallbeschreibung

Zusätzlich zu den aoA können optional noch textuelle Beschreibungen eines Anwendungsfalles erstellt werden. Sie sind folgendermaßen strukturiert:

Name , Akteure , VB, (Haupt- |Alternativer-) Fluss, NB

Anwendungsfallmodell

Ein Überblick über das Gesamtsystem ist alleine auf Grundlage der Aktivitätsdiagramme und textuellen Beschreibungen nicht möglich. Ein **Anwendungsfallmodell** ermöglicht insbesondere das **Include-** und **Extend-Beziehungsgeflecht** zu erkennen.

Für eine **Menge von Anwendungsfällen** erhält man das zugehörige **Anwendungsfallmodell**, indem man wie folgt vorgeht:

- ◆ Jeder **Anwendungsfall** wird übernommen
- ◆ Die **Akteure** werden übernommen und über eine Assoziation verbunden
- ◆ Eine **Include-Beziehung** wird verwendet, wenn ein Anwendungsfall einen anderen zur Ausübung seiner Hauptfunktionalität benötigt. Der **Übergeordnete** ist vom inkludierten Anwendungsfall abhängig, der **Inkludierte** besitzt keine Kenntnis vom Übergeordneten. Wichtig für **Wiederverwendung**.
- ◆ Eine **Extend-Beziehung** wird verwendet, wenn durch einen Anwendungsfall (**Erweiterungsanwendungsfall EAWF**) die Hauptfunktion eines anderen Anwendungsfalles (**Basisanwendungsfall BAWF**) erweitert wird (zB durch **zusätzliche Features**). BAWF ist unabhängig vom EAWF, der dagegen vom BAWF abhängt da er ihn und den Extension Point kennen muss.

Ist die Art der Beziehung fraglich, ist die einfachere Include-Beziehung vorzuziehen.

Navigation

Navigation ist jede Form von Benutzereingriffen in den Geschäftslogikablauf. Man unterscheidet zwei Arten von Navigation:

- ◆ **geschäftslogik-abhängige Navigation**: folgt stets dem Kontrollfluss des Anwendungsfalles. Spiegelt sich in den Aktivitätsdiagrammen in Form von Akteurentscheidungen wieder.

- ◆ **geschäftslogik-unabhängige Navigation:** Benutzer sollte AWF abbrechen oder neu starten sowie andere AWF aufrufen können. Wird durch **Links** unterstützt (Startseite, AWF-Auswahl, AWF-Neustart, optional Systemabmeldung).

Globales Bildschirm-Layout

Häufig verwendetes **Standard-Muster** für die **globale Layout-Struktur:**

- ◆ Statische **Kopfzeile** mit Firmeninformationen
- ◆ Statische **Fußzeile** mit Copyright-Informationen, Links zu Impressum/Datenschutzbestimmungen
- ◆ **Navigationsleiste** mit Links für Geschäftslogik-unabhängige Navigation.
- ◆ Der verbleibende Bereich für den eigentlichen Inhalt (Anwendungsfallbereich bzw **AWF-Bereich**)

Unterhalb der Kopfzeile kann noch eine **Orientierungszeile** eingefügt werden, die den Namen des gerade **aktiven AWF** sowie den Namen des **aktuellen Akteurknotens** enthält. Der AWF-Bereich kann sich beliebig weit nach unten verschieben („Papierrolle“) -> schlechte Ergonomie.

Benutzbarkeit

Überprüfung der anforderungsorientierten Aktivitäten hinsichtlich der Benutzbarkeit. Die Verbesserung der Benutzbarkeit erfolgt über die **Ergonomie der UI** sowie den **Zuschnitt der Arbeitsschritte** in den AWF. **Rahmenbedingungen** für das **Verbesserungspotential** werden durch Limitierung des einer Szene zur Verfügung stehenden AWF-Bereiches vorgegeben. Ist **Szene** in AWF-Bereich lesbar, oder muss sie **aufgeteilt** werden? Parallel dazu muss der zugehörige **Akteurknoten** in entsprechende einzelne Akteurknoten zerlegt werden.

Evtl können auch **kleine Szenen** (und zugehörige Akteurknoten) **zusammengelegt** werden. Dazu müssen 3 Bedingungen erfüllt sein:

- ◆ Szenen passen in einen AWF-Bereich und folgen im Kontrollfluss direkt aufeinander
- ◆ Szenen bzw Akteurknoten stehen in fachlichen Zusammenhang
- ◆ Die zusammengesetzte Aktivität muss semantisch äquivalent zur Ausgangsaktivität sein. Auch die zwischen den zusammengezogenen Akteuraktionen liegenden Systemaktionen werden zu einer einzigen zusammengefasst.

Das **Zusammenlegungsverfahren** kann **iteriert** werden. In unklaren Fällen sollte die Benutzbarkeit anhand von **Mockups** geprüft werden.

Kap25: Domänenklassenmodell

Zur Ermittlung der Domänenobjekte bzw. Domänenklassen werden **zwei Quellen** herangezogen:

- ◆ Produktdaten des **Lastenhefts** sowie Gegenstände und Sachverhalte des **Glossars**.
- ◆ Objekte bzw Daten auf die man während der **Ermittlung von Anwendungsfällen** und ihrer **Modellierung als aoA** stößt (typischerweise in Szenen). Sie werden mit Produkt-

daten und Glossareinträgen abgeglichen und geeignet ergänzt.

Die **Namen von Attributen** sollten in Szenen und im Domänenklassenmodell übereinstimmen.

Assoziationen und Generalisierungsbeziehungen zwischen Klassen werden aus den relevanten Beziehungen der Domänenobjekte gewonnen. Auch die Aktivitäten geben Hinweise.

Domänenklassen werden **nicht mit Operationen** versehen, um unerwünschte Realisierungsaspekte zu vermeiden. Das Modell ähnelt daher einem ER-Modell.

Alle **Anforderungsmodelle** sollen **problemadäquat** sein. Für das Domänenklassenmodell bedeutet dies, dass jedes Element einen relevanten Gegenstand oder Sachverhalt als Pendant in der Problemwelt haben muss. Insbesondere **Generalisierungsbeziehungen** sind in der Realwelt **nur selten** anzutreffen. Ausnahmen von dieser Regel sind zu hinterfragen und zu begründen.

KE 7

Teil IX Softwarespezifikation

Kap27: Ziele und Vorgehensweise

Die **Softwarespezifikation** von **ABEweb** bereitet die Anforderungsspezifikation dahingehend auf, dass sie als **Ausgangsbasis** für eine **systematische Transformation** in das **Klassenmodell des Softwareentwurfs** verwendet werden kann.

In erster Linie werden die aoA in ssoA überführt (**ssoA werden aus aoA generiert**). Dabei werden **zusätzliche Systemaktionen** eingefügt und die **UML-Verträglichkeit** wiederhergestellt. Die Einzelschritte der Transformation sind rein technischer Natur und können automatisiert werden. Der Vorteil von ssoA ist die besonders **einfache Abbildung** in das **Entwurfsklassenmodell**.

Systemaktivitäten werden ebenfalls **UML-konform** gemacht und evtl durch zusätzliche Angabe von Objekten und Objektflüssen **weiter präzisiert**.

Das **Domänenklassenmodell** bleibt idR unverändert. Bei der Überarbeitung der Systemaktivitäten neu identifizierte Objekttypen sind natürlich aufzunehmen (wodurch man ein **Klassenmodell der Softwarespezifikation** erhält).

Dokumente der Softwarespezifikation:

- ◆ sw-spezifikationsorientierte Aktivitäten (ssoA)
- ◆ Systemaktivitäten
- ◆ Domänenklassenmodell

- ◆ ergänzende Modelle und Beschreibungen aus der Anforderungsspezifikation.

Kap28: Transformation Anforderungs- zu Softwarespezifikation

Transformation der anforderungsorientierten Aktivitäten (aoA)

Ziele der Transformation sind:

- ◆ Jede **Kante**, die von **Akteurknoten** ausgeht, soll in **Systemknoten** enden (da bei interaktiven Systemen auf jede Benutzeraktion eine Systemreaktion folgt).
- ◆ Die **UML-Verträglichkeit** wird hergestellt.

Die Transformation **aoAktivität** -> **ssoAktivität** geschieht in **4 Schritten**:

- ◆ **Schritt 1: Systemreaktion**

Endet eine von einem Akteurknoten ausgehende Kante nicht in einem Systemknoten, wird eine **Systemaktion eingefügt**.

- ◆ **Schritt 2: Akteurknoten mit systemgefüllten Attributen** in einer Szene

Angezeigte oder zu editierende Attribute in Szene sind systemgefüllt. Der dem zur Szene gehörenden Akteurknoten unmittelbar vorausgehende Systemknoten übernimmt idR diese Aufgabe und greift dafür auf Anwendungslogik zu. Gibt es einen unmittelbaren Vorgängerknoten der kein Systemknoten ist (ohne Berücksichtigung von Zusammenführungsknoten) wird eine **Systemaktion eingeschoben**, die die Beschaffung des dynamischen Inhalts übernimmt.

- ◆ **Schritt 3: AWF-Subaktivität**

Eine AWF-Subaktivität wird gewöhnlich innerhalb verschiedener Aktivitäten wieder verwendet, wobei sie ihre jeweilige übergeordnete Aktivität nicht kennt. Sie kennt daher auch nicht den Anschlussknoten in der übergeordneten Aktivität. Um den weiteren **Verlauf des Kontrollflusses** sicherzustellen, wird ein **expliziter Stack** angelegt, auf dem der **Anschlussknoten** abgelegt wird. Die AWF-Subaktivität nimmt nach Beendigung diese Information vom Stack und gibt so die Kontrolle an die übergeordnete Aktivität zurück. Gibt es vor der Subaktivität keinen **Systemknoten** der den Anschlussknoten auf den Stack legen kann, wird dieser **eingefügt** (analog zu Schritt 2).

- ◆ **Schritt 4: UML-Verträglichkeit**

Um die UML-Verträglichkeit aoA wieder herzustellen, müssen lediglich die **Entscheidungen** UML-konform gemacht werden, die als **Aktionen** (<<AE>> oder <<SE>>) mit **meheren Ausgängen** modelliert wurden. Sie werden wieder in eine **Aktion** (<<AA>> bzw <<SA>>) gefolgt von einem **Entscheidungsknoten** aufgeteilt.

Transformation der Systemaktivitäten

Systemaktivitäten werden wie die aoA **UML-konform** gemacht. Ist Anwendungslogik einer Systemaktivität hinreichend komplex, kann die zusätzliche Angabe von **Objekten** und **Objektflüssen** sinnvoll sein.

Teil X Softwarearchitektur

Kap30: Softwarearchitektur

Die eigentliche **Realisierung des Softwaresystems** beginnt mit der Konzeption der **Softwarearchitektur**, bei der die zentralen Realisierungsentscheidungen getroffen werden. Die Architektur legt das grundlegende **Strukturierungsschema** der Software und das Zusammenspiel der strukturbildenden Bestandteile (**Architekturkomponenten**) fest. Die Wahl der Architektur wird besonders durch die **nicht-funktionalen Anforderungen** bestimmt. Sie bildet die **feste Rahmenvorgabe**, der sich die weitere Realisierung (insbesondere der Softwareentwurf) unterordnen müssen.

Die Überlagerung der **5-Schichten-Architektur** mit dem **MVC- Model** ergibt die vier Kernkomponenten der **ABEweb-Architektur**:

- ◆ View/Controller, Anwendungslogik, Anwendungsobjekte.

Auf diese noch plattformunabhängige Grobarchitektur werden zwei Verfeinerungsschritte angewandt:

(1)

Eine an den Anwendungsfällen orientierte Zerlegung. **Controller**, **View** und **Anwendungslogik** werden die zu realisierenden Komponenten eines jeden AWFs jeweils als lokale Komponente zugeteilt. Jede der drei Architekturkomponenten enthält noch zwei weitere lokale Komponenten:

- ◆ Bündelung aller Artefakte, die nicht ausschliesslich einem Anwendungsfall zugeordnet werden können.
- ◆ Bündelung aller AWF-übergreifenden Artefakte

Anwendungsobjekte werden nicht weiter zerlegt, da sie häufig in mehreren AWF vorkommen.

Ergebnis dieses Schritts: **plattformunabhängige Feinarchitektur**

(2)

Einbeziehung der **plattformspezifischen Eigenschaften** (Struts und Java EE). Die Architekturkomponenten werden als **Java-Pakete** realisiert:

- ◆ **MVC-Controller** <<**Paket**>>_C: nimmt Struts-Komponenten ActionForms und Actions auf sowie alle übrigen zum Controller gehörenden Hilfsklassen. Für jede ssoA (jeden AWF) wird ein Unterpaket für die zugeordneten Struts-Komponenten hinzugefügt, sowie die zusätzlichen Unterpakete „gemeinsameAwfKomponenten“ und „awfUebergreifendeKomponenten“.
- ◆ **MVC-View** <<**Verzeichnis**>>_V: Als Textdateien werden JSP's in Verzeichnisstrukturen angeordnet. Für jede ssoA (jeden AWF) wird ein Unterpaket für die zugeordneten JSPs hinzugefügt, sowie die zusätzlichen Unterpakete „gemeinsameAwfKomponenten“ und „awfUebergreifendeKomponenten“.
- ◆ **MVC-Model** <<**Paket**>>_Logik, [**Paket**>>_Objekte]: nehmen Session-Beans [bzw Entity-Klassen] und Hilfsklassen auf. Für jede ssoA (jeden AWF) wird ein Unterpaket hinzugefügt, sowie die zusätzlichen Unterpakete „gemeinsameAWFKomponenten“ und „awfUebergreifendeKomponenten“.

Ergebnis dieses Schritts: **plattformspezifische Feinarchitektur = ABEweb-Softwarear-**

chitektur.

Teil XI Softwareentwurf

Kap31: Ziele und Vorgehensweisen

Ausgangspunkt für den **Softwareentwurf** bilden die **Softwarespezifikation** (funktionale Anforderungen) und die **ABEweb-Softwarearchitektur** (strukturelle Rahmenbedingungen). Gesteuert von Softwarespezifikation verfeinern (architekturkonforme lokale) Entwurfsentscheidungen die Architektur in kleinere beherrschbare Einheiten (zB Klassen und Pakete -> Komplexitätbewältigung). Nichtfunktionale Anforderungen müssen am Ende des Entwurfs erfüllt sein.

ABEweb-Ansatz: die zentralen Bestandteile der Softwarespezifikation in 4 Schritten systematisch in ein architekturkonformes Entwurfsklassenmodell transformieren:

- ◆ Domänenklassenmodell als Ausgangsbasis für das Entwurfsklassenmodell übernehmen.
- ◆ ssoA in das Entwurfsklassenmodell transformieren
- ◆ Systemtätigkeiten in das Entwurfsklassenmodell abbilden
- ◆ Verfeinerung „im klassischen Stil“ des aus den Transformationen hervorgegangenen Entwurfsklassenmodells, insb Vervollständigung des Entwurfs der Anwendungslogik.

Die **Transformation der Softwarespezifikation** lässt sich automatisieren, so dass weite Teile des Entwurfsklassenmodells generiert werden können. Die Transformation der **ssoA** ist einfach, da Knoten und Kanten der Aktivitäten 1:1 auf Klassen, Operationen und Abhängigkeiten (dependencies) des Entwurfsklassenmodells abgebildet werden können. Eine **Szene** kann auf bis zu 3 Klassen abgebildet werden, unter denen genau eine JSP vorkommt.

Die **Transformation der Systemaktivitäten** ist ebenfalls eine 1:1 Abbildung.

Das **resultierende Entwurfsklassenmodell** sollte eine **Detaillierung** aufweisen, die direkte Implementierung erlaubt.

Die Modellierung mündet in eine **Entwurfsspezifikation:**

- ◆ das architekturkonforme Entwurfsklassenmodell
- ◆ ggf Interaktionsmodelle (Sequenz- und Kommunikationsmodelle)

Kap32: Transformation Domänen- in Entwurfsklassenmodell

Im ersten Entwurfsschritt wird das **Domänenklassenmodell** unverändert übernommen. Es orientierte sich bisher an der **Problemadäquatheit** und wird jetzt in Bezug auf Erfordernisse der **Realisierung** überarbeitet (insb auf weitere mögliche **Generalisierungen** untersucht). Dann wird geprüft, welche Klassen **persistiert** werden sollen und welche nicht (Stereotypen <<Entity>> und <<transient>>).

Kap33: Transformation der ssoA in das Entwurfsklassenmodell

Transformationsregeln

Im zweiten Entwurfsschritt wird jede ssoA wie folgt transformiert:

◆ Regel 1: Aktivität

Stateless Session Bean-Klasse anlegen, die Anwendungslogik des AWF kapselt.

awsName_Logik.unterpaketAnwendungsfall.NameAnwendungsfallAL

mit `<<Stateless Session Bean>>` stereotypisiert

◆ Regel 2: Akteuraktion

In DispatchAction | LookupDispatchAction abgebildet (nur Links | nur Buttons als Entscheidungsmöglichkeiten).

awsName_C.unterpaketAnwendungsfall.NameAkteuraktion(DA|LDA)

mit `<<DispatchAction|LookupDispatchAction >>` stereotypisiert

wenn Links und Buttons vorkommen -> DA (für Links) und LDA (für Buttons)

◆ Regel 3: Szene

In JSP abgebildet.

awsName_V.unterpaketAnwendungsfall.NameAnwendungsfall

mit `<<Java Server Page>>` stereotypisiert

◆ Regel 4: Szene mit Benutzereingaben

ActionForm AF anlegen für Szenen mit `<<einzugebenden|zu editierenden Attribute>>` oder `<<(selektierbare) Listen>>`.

awsName_C.unterpaketAnwendungsfall.NameAkteuraktionAF

mit `<<ActionForm>>` stereotypisiert

Aus Software Engineering (SE)-Sicht: bei mehreren Szenen pro AWF und vielen AWF im System kann Anzahl AF und Attributredundanz groß werden. Dem Zusammenfassen von AF bzw der Oberklassenbildung steht aber die geringere Kohäsion bzgl eines Request sowie die evtl große Anzahl für einen Request nicht benötigter Attribute entgegen. Diese können von User manipuliert werden (über Browser-Adresszeile)

◆ Regel 5: Szene mit vom System angezeigten Attributwerten

Von Struts unabhängiges Data Transfer Object-Klasse (DTO) anlegen für Szenen mit `<<angezeigte Attribute>>` oder `<<Listen>>`.

awsName_C.unterpaketAnwendungsfall.NameAkteuraktionDTO

mit `<<Data Transfer Object>>` stereotypisiert

Aus Software Engineering (SE)-Sicht: Kopplung zwischen View V und Model M ist relativ lose, da Anwendungslogik AL alle in Szene benötigten Attribute als (komplette) Kopien der betroffenen Entities dem Controller C übergibt. C stellt dann nur die tatsächlich anzuzeigenden Attribute in DTO sowie die zu editierenden Attribute in AF zusammen und stellt sie V über Scope zur Verfügung. Wird weiteres Attribut benötigt -> nur Webschicht JSP und DTO betroffen. Nachteil: evtl zu viele DTO-Klassen, erhöhter Datentransport, reduzierte Sicherheit.

Die DTOs können auch ganz eliminiert werden -> AF nimmt alle Attribute auf, auch

die anzuzeigenden (aber nur mit Getter-Methoden). Vorteil: Anzahl Klassen in C erheblich reduziert. Nachteil: mehr Abhängigkeit von Struts, der Name des AF muss beim Zugriff auf anzuzeigende Attribute mittels EL in JSP bekannt sein (-> normalerweise transparent).

Andere Variante: DTO für Datentransport zwischen Model (AL) und Webschicht (C). C nimmt DTO entgegen -> zu editierende Attribute in AF, anzuzeigende Attribute in (derselben) DTO über Scope an V. Vorteil: reduzierter Datentransport, erhöhte Sicherheit. Nachteil: evtl viele DTOs, mehr Kopplung zwischen Webschicht und M. Neues Attribut -> Webschicht und AL ändern.

- ◆ **Regel 6: Systemaktion mit Anwendungslogik**

Für jede Systemaktion sowie für das Initialisieren und Beenden des AWF wird (sofern Anwendungslogik involviert ist) eine Operation zur Stateless Session Bean (Regel 1) hinzugefügt.

Aus Software Engineering (SE)-Sicht: auch für SA, in die Anwendungslogik involviert ist -> nur eine einzige OP in Stateless Session Bean-Klasse, die von zugehöriger Action aufgerufen wird. Die Action ruft also nur eine einzige OP auf und wird nicht durch die Koordination der Aufrufe verschiedener AL-OP mit AL verschmutzt -> Separation of Concerns (Webschicht bzw C <-> Model bzw AL). Wird AL-OP zu komplex -> Hilfsklassen einführen. Besitzt SA 2 Nachfolgeaktionen A1 und A2 abhängig von Bedingung B, dann sollte allein die AL-OP feststellen ob B erfüllt ist und die Information zusammen mit den Rückgabewerten an Action-OP zurückliefern. Die Action-OP vollzieht lediglich die Entscheidung, indem sie die Weiterleitung anstößt.

- ◆ **Regel 7: Systemaktion mit vorangehender Akteuraktion**

Auf Operation in zugeordneter (L)DA abgebildet. Sind AA 2 Klassen zugeordnet -> Operation zu der Klasse, die Funktionalität des gedrückten Link bzw Button bereitstellt. Operation verarbeitet Benutzeranfrage, für Anwendungslogik wird OP aus Regel 6 aufgerufen.

- ◆ **Regel 8: Systemaktion ohne vorangehende Akteuraktion**

Auf „normale“ Action abgebildet.

awsName_C.unterpaketAnwendungsfall.NameSystemaktionA

mit <<Action>> stereotypisiert

- ◆ **Regel 9: Startknoten**

Auf „normale“ Action abgebildet. Ist Einstiegspunkt für AWF (kapselt ihn) -> prüft VB, erledigt Initialisierungsarbeit

awsName_C.unterpaketAnwendungsfall.StartNameAnwendungsfallA

mit <<Action>> stereotypisiert

- ◆ **Regel 10: Endknoten**

Für alle Endknoten eine gemeinsame DA. Für die Endknoten des Main Flow und der Alternative Flows -> eine gemeinsame OP, für jeden Exceptional Flow -> jeweils eine OP. OP sind Abschlusspunkte des AWF -> prüfen NB, erledigen Aufräumarbeiten oder spezielle Fehlerbehandlungen. Ist AWF in anderen AWF eingebettet -> OP nimmt oberstes Element vom Stack (Kontrolle zurückgeben). Für Anwendungslogik wird OP aus Regel 6 aufgerufen.

awsName_C.unterpaketAnwendungsfall.EndNameAnwendungsfallIDA

mit <<Action>> stereotypisiert

◆ **Regel 11: Extend-Beziehung**

Ist AWF an Extend-Beziehung beteiligt (egal ob BasisAWF oder ErweiterungsAWF) wird von Struts unabhängige JavaBean-Klasse zur Verwaltung dieser Beziehung erstellt die für jeden Extension Point ein Attribut bekommt, welches den ErweiterungsAWF (zB URL StartAction) aufnimmt.

ErweiterungsAWF: OP die beim Applikationsstart aufgerufen wird und AWF beim Extension Point des BasisAWF registriert (zB URL StartAction in Attribut ablegt).

awsName_C.unterpaketAnwendungsfall.JavaBeanName

mit <<Extend-Verwaltung>> stereotypisiert

In jeder Systemaktion SA sollten vor Ausführung der eigentlichen Funktionalität die VB für SA geprüft werden (User authorisiert? Attributwerte vorhanden und in gültigen Wertebereich?), da SA direkt über Browseraddresszeile angesprochen werden können. Dies gilt nicht für Akteuraktionen AA, die als JSP innerhalb WEB-INF-Verzeichnis abgelegt sind.

AWF-Subaktivitäten werden als eigenständige Aktivitäten transformiert, nicht innerhalb übergeordneter Aktivität.

<i>Regel</i>	<i>Element aus ssoA</i>		<i>Element aus Entwurfsklassenmodell</i>
<i>Regel 1</i>	Aktivität	=>	Stateless Session Bean
<i>Regel 2</i>	Akteuraktion	=>	(Lookup)DispatchAction
<i>Regel 3</i>	Szene	=>	JavaServerPage
<i>Regel 4</i>	Szene mit Benutzereingaben	=>	ActionForm
<i>Regel 5</i>	Szene mit vom System angezeigten Attributwerten	=>	DTO-Klasse
<i>Regel 6</i>	Systemaktion mit Anwendungslogik	=>	Operation in Stateless Session Bean
<i>Regel 7</i>	Systemaktion mit vorangehender Akteuraktion	=>	Operation in der (Lookup)DispatchAction
<i>Regel 8</i>	Systemaktion ohne vorangehende Akteuraktion	=>	„normale“ Action
<i>Regel 9</i>	Startknoten	=>	„normale“ Action
<i>Regel 10</i>	Endknoten	=>	DispatchAction mit Operationen
<i>Regel 11</i>	Extend-Beziehung	=>	Extend-Verwaltungsklasse

Entwurfsklassenmodell

Die aus der Transformation des Domänenklassenmodells und der ssoA entstehenden Entwurfsklassen sind allerdings noch völlig **isoliert**, denn es fehlen die Beziehungen. Die relevanten Beziehungen sind **Abhängigkeiten (dependencies)**, die unterschiedliche **Benutzungsbeziehungen** darstellen.

◆ <<use>>-Abhängigkeit

OP der Klasse A ruft OP der Klasse B auf (Action-Klasse -> Stateless Session Bean, Stateless Session Bean -> Entity-Klasse)

◆ **<<get>>-Abhängigkeit, <<set>>-Abhängigkeit**

<i>Klasse A</i>	<i>Klasse B</i>	<i>Spezielle Form <<use>></i>
Action	Action-Form	<<get>>, <<set>>
Action	DTO	<<get>>, <<set>>
Action	Extend-Verwaltungsklasse	<<get>>
Extend-Verwaltungsklasse (Erweiterungs-AWF)	Extend-Verwaltungsklasse (Basis-AWF)	<<set>>

◆ **<<forward>>-Abhängigkeit**

Nach Abschluss jeder OP einer Action wird an JSP oder nächste Action-OP weitergeleitet. Indirekt über Framework/Konfigurationsdatei realisierte Abhängigkeit Action -> JSP bzw Action -> Action.

◆ **<<forwardByInclude>>-Abhängigkeit**

Wenn Systemaktion eine AWF-Subaktivität aufruft, legt sie die nachfolgende Action auf den Stack: Abhängigkeit Systemaktion -> Action.

◆ **<<request>>-Abhängigkeit**

Indirekte Abhängigkeit einer JSP zu einer Action, indem mittels einer von der JSP generierten HTML-Seite über einen Request eine Action angesteuert wird.

◆ **<<setByRequest>>-Abhängigkeit**

Indirekte Abhängigkeit einer JSP zu einer Action Form, indem mittels einer von der JSP generierten HTML-Seite (Eingabe-) Daten an den Server gesendet werden und diese vom Framework in einer ActionForm abgelegt werden.

Kap35: Exkurs Transformationsvariante

Die obigen Transformationsregeln haben **2 Nachteile**:

- ◆ relativ viele Action-Klassen, die nur durch Paketbildung AWF-bezogen zusammengefaßt werden können.
- ◆ Die Entscheidung zwischen DA und LDA hängt davon ab, ob Buttons oder Links benutzt werden -> reine Oberflächenentscheidung, wird spät getroffen, kann sich jederzeit ändern

Lösung für beide Probleme:

eine **eigene abstrakte Action** erstellen, die **alle Action-OPs** für eine **Aktivität** aufnimmt sowie **Links und Buttons** gleichermaßen **verarbeiten kann**. Das Weiterleiten auf eine andere Action-OP muß möglich sein. Die abstrakte **UseCaseAction** des Lehrgebietes ermöglicht es, nur noch eine einzige konkrete Action pro Aktivität zu erstellen, die nicht zwischen Buttons und Links unterscheidet und den uneingeschränkten Einsatz von Subaktivitäten ermöglicht.

Die Transformation von **Szenen** bleibt unverändert. Statt einer (L)DA pro **Akteuraktion** wird jetzt eine UseCaseAction pro ssoA erstellt. Es werden nur noch **Systemaktionen**

transformiert, Akteuraktionen nicht mehr. Der UseCaseAction wird für jede Systemaktion eine OP hinzugefügt.

Kap36: Transformation Systemaktivitäten in Entwurfsklassenmodell

Der dritte Entwurfsschritt von ABWeb besteht darin, jede Systemaktivität in das Entwurfsklassenmodell (in eine Stateless Session Bean) zu transformieren.

◆ Regel 12: Systemaktivität

Für die SA wird eine Stateless Session Bean erstellt, welche für die Implementierung der in der SA enthaltenen Aktionen verantwortlich ist.

awsName_Logik.unterpaketAnwendungsfall.NameSystemaktivität

mit <<Stateless Session Bean>> stereotypisiert

◆ Regel 13: Aktionen

Für jede Aktion der SA wird der Klasse aus Regel 12 eine OP mit dem Namen der Aktion hinzugefügt. Die hinzugefügte OP wird von der OP aufgerufen, die auf der nächsthöheren Hierarchieebene für Aktion erzeugt wurde.

Pro SA in ssoA wird maximal 1 OP in der Stateless Session Bean der obersten Hierarchiestufe vorgesehen. Dadurch ruft OP der zugehörigen Struts-Action nur eine einzige OP der AL auf und nimmt deren Ergebnis entgegen. Sie wird also nicht durch Koordination der Aufrufe mehrerer AL-OPs mit AL verschmutzt. C (Webschicht) und M (AL) bleiben klar getrennt (Separations of Concerns). Konzept entspricht dem Entwurfsmuster Session Fassade.

Kap37: Verfeinerungen und Ergänzungen des Entwurfsklassenmodells

Allgemeines

Im abschliessenden 4. Entwurfsschritt wird das bisher aus Transformationen hervorgegangene Entwurfsklassenmodell „im konventionellen Stil“ verfeinert und ergänzt. Den Schwerpunkt bildet die **Vervollständigung des Entwurfs der Anwendungslogik**, der von den Transformationen nur rudimentär erzeugt wurde. Ziel ist ein Entwurfsklassenmodell, das eine Detaillierung aufweist, die eine direkte Implementierung ermöglicht.

Vervollständigung des Entwurfs der Anwendungslogik

Pro AWF existiert (wenn SA fehlen) lediglich eine Stateless Session Bean, deren OPs unvollständige Signaturen haben. Entities sind noch ohne OPs.

◆ Vervollständigung der OP-Signaturen von Stateless Session Beans

Nur OPs die über Regel 6 und 13 erzeugt wurden. Werden neue Objekttypen eingesetzt (zB Enumerationsklassen oder DTO-Klassen), sind diese dem Entwurfsklassenmodell hinzuzufügen.

◆ Ausstattung der Entities mit OPs

Für jede OP einer Stateless Session Bean wird geprüft (zB über Interaktionsdia-

gramme), ob sie eine OP einer Entity zur Erfüllung ihrer Aufgabe benötigt -> Entity OP mit Parametern hinzufügen. Die OP's einer Entity sollten sich nach Möglichkeit nur auf der Entity selbst und evtl noch auf eng assoziierten Entities (zB Kompositionsbeziehung) abstützen.

Da jetzt die OP's für alle Entwurfsklassen feststehen, ist das Ablaufverhalten auf der Ebene von Operationen determiniert. Daher kann jetzt die Navigierbarkeit von Assoziationen (zwischen Entities) präzisiert werden.

◆ Interaktionsmodelle

Entwurfsschritt 4 weist einen Detaillierungsgrad auf, der Interaktionsmodelle (Sequenz- und Kommunikationsmodelle) zu einem adäquaten Hilfsmittel für die Veranschaulichung ablaufforientierten Verhaltens macht. Die relativ realisierungsunabhängigen Aktivitäten sind nicht geeignet, für ein Szenario die Reihenfolge der OP-Aufrufe inklusive der aktuellen Parameter auszudrücken.

Sonstige Verfeinerungen und Ergänzungen

Einige **Modellierungselemente** der UML (zB Assoziationen unterschiedlicher Multiplizität und Navigierbarkeit, Assoziationsklassen) müssen noch **umgeformt** werden, um in Java kanonisch implementiert werden zu können.

Für **Stateless Session Bean-OP's** mit länger andauernden, nicht notwendigerweise synchron auszuführenden Aufgaben bietet es sich an zusätzlich eine **Message Driven Bean** einzusetzen, welche die eigentliche Aufgabe übernimmt.

[Stand WS 07/08]