Open Source Webentwicklungsframeworks und Webanwendungsentwicklung mit SAP Software am Beispiel einer Referenzanwendung

Diplomarbeit

Prof. Dr. Wolfram Reiners
Dipl.-Ing.(FH), Dipl.-Inf.-Wiss Diemar Mogwitz

Fachhochschule Furtwangen
Hochschule für Technik und Wirtschaft

vorgelegt von Simon Pepping WI BN 8
Furtwangen, 28.02.2005

Inhaltsverzeichnis

Inhaltsverzeichnis ... 2
Abstract ... 4
1 Jakarta Struts ... 6
1.1 Kapitelübersicht ... 7
1.2 Grundstrukturen und Basiskomponenten ... 8
1.3 Die Initialisierung einer Struts Webapplikation ... 16
1.4 Das Request Processing ... 28
1.5 Zusammenfassung ... 35
2 Sun Java Server Faces ... 36
2.1 Kapitelübersicht ... 36
2.2 Grundstrukturen und Basiskomponenten ... 38
2.2.1 Grundlegende Klassen der Java Server Faces ... 39
2.2.2 Der Request Processing Lifecycle ... 44
2.2.3 Das GUI Komponentenkonzept und Rendering ... 49
2.2.4 Event gesteuertes Aktion Handling – JSF Strategie des MVC Musters ... 50
2.3 Die Initialisierung einer JSF Webapplikation ... 53
2.4 Das Request Processing ... 68
2.5 Zusammenfassung ... 82
3 Komponenten und Strukturen des SAP CRM ... 83
3.1 Kapitelübersicht ... 83
3.2 Datenstrukturen und Beziehungen von SAP CRM Komponenten ... 84
3.2.1 Aufbau und Struktur von SAP Datenbanktabellen ... 84
3.2.2 Häufig verwendete Fachbegriffe ... 86
3.3 Der Geschäftspartner ... 88
3.3.1 Umgebungstabellen des Geschäftspartners ... 91
3.3.2 Customizing - Tabellen des Geschäftspartners ... 96
3.4 Der Geschäftsvorgang ... 102
3.4.1 Extensiontabellen des Geschäftsvorgangs und -positionen ... 106
3.4.2 Set Tabellen des Geschäftsvorgangs und dessen Positionen ... 109
3.4.3 Customizing Tabellen des Geschäftsvorgangs und dessen Positionen ... 117
3.5 Zusammenfassung ... 123
4 SAP Web Dynpro ... 124
4.1 Kapitelübersicht ... 124
4.2 Design und Konzeption einer SAP Web Dynpro Referenzanwendung ... 125
4.2.1 Features und Funktionalitäten der Referenzanwendung ... 126
4.2.2 Die Reihenfolge des Web Dynpro Entwicklungsprozesses ... 126
4.2.3 Die Ergebnisse der Modelcodegenerierung ... 128
4.2.4 Der Controller und View Context ... 131
4.2.5 Das View Design und das Navigation Handling ... 132
4.2.6 Grundlegende Codeimplementierung ... 133
4.3 Der Aufbau des Frameworks ... 135
4.4 Die Modelkomponente ... 135
4.4.1 Die Hierarchie bis Z_All_Leads_By_Partner_2_Input ... 136
4.4.2 Die Hierarchie bis `LeadModel` ... 138
4.5 Die Controllerkomponente ... 141

4.5.1 Die Basis: Context, Controller und Component ... 141
4.5.2 Der Context im Detail: Knoten und Elemente ... 143
4.5.3 Die Umsetzung: Generierte Klassen mit Controllerfunktionalität ... 144
4.6 Die Viewkomponente ... 146
4.6.1 Die Basisschnittstellen der Viewkomponente ... 146
4.6.2 Derivate der View Klasse ... 146
4.6.3 Die Umsetzung: Generierte Klassen mit Viewfunktionalität ... 148
4.7 Die Web Dynpro UI Komponentenbibliothek ... 149
4.8 Zusammenfassung ... 150
5 Anhang ... 151
A Das Struts Framework in Kurzform ... 151
B Das JSF Framework in Kurzform ... 153
C Konfiguration der Verbindung von SAP System und SAP Web Application Server 154
D Screenshots der Klassendiagramme zum Aufbau des Web Dynpro Frameworks ... 156
6 Literaturverzeichnis ... 177
6.1 Internetressourcen ... 177
6.2 Literatur ... 178
6.3 Abbildungsverzeichnis ... 179
6.4 Tabellenverzeichnis ... 181

Abstract

Frameworks sind aus der heutigen Softwareentwicklung nicht mehr weg zu denken. Bauten Softwareentwickler früher noch selbst Grundstrukturen auf, um ihre Programme übersichtlicher, und leichter wartbar zu machen, und um ihre eigene Programmiertätigkeit durch den Wegfall von sich immer wiederholenden Aufgaben zu beschleunigen, werden Standardentwicklungstätigkeiten heute von Frameworks erledigt. Dies ermöglicht eine Konzentration auf das Wesentliche der Anforderungen eines Softwareentwicklungsprojekts.

Heute sind Frameworks standardisiert, im Falle von Open Source, in der Nutzung und durch eigene Beiträge jedermann zugänglich und unterliegen einem permanenten Weiterentwicklungsprozess. Die Weiterentwicklung kann sowohl Verbesserung und kleinere Erweiterungen des Bestehenden umfassen bis hin zur Neustrukturierung aufgrund von technischen Änderungen und Vorraussetzungen oder gar neuen Standards, die eine komplette Überarbeitung erfordern.

Die heutigen Frameworks bieten Entwicklern für Basisaufgaben ihre Anwendungsprogrammierung wie z.B. Webapplikationen oder verteilte komplexe J2EE Backendanwendungen mit Netzwerkinfrastrukturen, ein einheitliches Programmiermodell und einheitlichen Zugriff auf benötigte Services ohne Kenntnisse haben müssen, wie das Framework intern arbeitet. Die Nutzung ist also vergleichsweise einfach zu erlernen, da die Komplexität der Funktionsweise gekapselt wird, und damit vollkommen transparent ist. Das Erlernen wird zudem durch umfachreiche Litaratur in Form von Büchern, eBooks und Zeitschriften erleichert, wenn ein Framework aufgrund seiner Bewährung in der Praxis, einmal genug Bekanntheit erreicht hat.

Um die Komplexität der Funktionsweise von Frameworks wieder in das Bewusstsein zu holen, und dem bisher nur nutzenden Entwickler neue Erkenntnisse und Einblicke zu vermittlen, die möglicherweise für seine eigenen Fähigkeiten und sein eigenes Wissen von Nutzen sind. Gelingt dies dem Leser mit Hilfe dieser Diplomarbeit, hat diese ihren maximalen Zweck erfüllt. Es ist es ein Ziel dieser Diplomarbeit anhand der zwei populären Open Source Webentwicklungsframeworks Jakarta Struts und den Java Server Faces von Sun, aufzudecken wie genau die Services und das Programmiermodell funktionieren, um das bekannte Entwicklungsschema bereitzustellen.

Bei der Firma SAP hat sich besonders im Bereich Java und dem Ziel, einen Zugriff auf SAP R/3 Systeme über eine ergonomischere Weboberfläche zu ermöglichen, in der letzten Zeit einiges getan. So gibt es seit letztem Jahr erstmals das Webframework Web Dynpro produktiv zu benutzen, das es ermöglicht Webanwendungen zu erstellen, die auf Daten eines SAP Systems zugreifen und diese darstellen, ohne die SAP GUI benutzen zu müssen.

Da es sich bei Web Dynpro ebenfalls um ein Framework handelt, liegt es im Bereich dieser Diplomarbeit, auch dessen Struktur und Funktionieren zu untersuchen. Da es zu diesem Thema bisher kaum Literatur gibt, ist es ein weiteres Ziel dieser Arbeit, dem Entwickler eine Anleitung an die Hand zu geben, mit der er wesentlich schneller und frustfrei zu Ergebnissen kommt. Ist diese Vertauens- und Wissensbasis geschaffen, kann er auch hier mit einem tieferen Einblick in die Internas von Web Dynpro fortfahren und selbst Vergleiche zu Struts und JSF anstellen.

Die wirtschaftlich verwertbare Programmierung mit Web Dynpro setzt je nach Themengebiet oder SAP Model Kenntnisse in ABAP und dem jeweiligen betriebswirtschaftlichen Umfeld voraus. Das dritte Ziel dieser Ausarbeitung ist demnach, an einem betriebswirtschaftlichen Brennpunkt, dem Customer Relationship Management, dessen datenmodelltechnische Umsetzung innerhalb der gegenwärtig aktuellen Version SAP CRM 4.0 zu erleuchten. Dies wird anhand

der CRM Geschäftsobjekte, bzw. Entitäten Geschäftspartner und Geschäftsvorgang durchgeführt.

Das große Finale besteht bei dieser Diplomarbeit darin, mit den Kenntnissen der Frameworks Struts, JSF und Webdynpro, dem Wissen über den SAP CRM Geschäftspartner und Geschäftsvorgang und der Praxis der Anwendungsentwicklung mit Web Dynpro, eine Referenzanwendung zu erstellen, die einen möglichen Anwendungsfall demonstiert, wie mit den beiden genannten Geschäftsobjekten gearbeitet werden kann.

1 Jakarta Struts

Der Web Entwicklungsframework Struts ist bis heute das bekannteste und am häufigsten verwendete Open Source Webframework in der Javawelt. Daher gibt es umfangreiches Literaturmaterial, das dem Entwickler schnellen Einblick in die Praxis der Programmierung mit Struts gewährt und es ihm ermöglicht, schnell und mit wenig Aufwand selbst Anwendungen zu erstellen. Mit der Zeit und mit wachsender Erfahrung bekommt der Entwickler eine Routine, wie sich auch komplexe Anforderungen mit Struts realisieren lassen. Durch die Standardisierung der Entiwcklung mit Struts werden Anwendungen besser wartbar und pflegbar und dies macht es für andere Programmierer leicht, schnell in ein Softwareentwicklungsprojekt einzusteigen.

Aufgrund der Masse an Literatur ist es nicht das Ziel dieses Kapitels eine weitere Programmieranleitung auszuarbeiten, sondern Struts unter einem Blickwinkel zu beleuchten, den es selten in Büchern oder Artikeln gibt. Das Ziel ist, dem Programmierer offenzulegen, wie Struts genau funktioniert. Dazu wird der Quelltext von Struts 1.1 analysiert und erklärt, was und wie etwas geschieht und welchem Zweck es dient. Wenn der Leser nach Durcharbeiten des Quelltextes mit Hilfe dieses Kapitels die Funktionsweise von Struts verstanden hat, und seine eigene Programmierweise mit diesem Wissen verbessert, hat dieses Kapitel seinen Zweck erfüllt.

Um einen Blick hinter die Kulissen von Struts werfen zu können, benötigt der Leser fortgeschrittene Java Programmiererfahrung. Sehr hilfreich ist auch die Erstellung von kleineren Webapplikationen mit Struts oder anderer Frameworks, denen das Servlet API zugrunde liegt.

1.1 Kapitelübersicht

Es gibt mittlerweile mehrere Frameworks, die die Entwicklung von Webanwendungen standardisieren und erleichtern. Grundlegende technische Elemente der Funktionsweise und Standardaufgaben von Webanwendungen werden gekapselt und es werden Richtlinien vorgegeben, die der Softwareentwickler einhalten muss, um mit den gebotenen Services die gewünschten Ergebnisse zu erhalten. Zu diesen technischen Funktionsweisen gehören:

- Die Umsetzung des Model-View-Controller Musters.

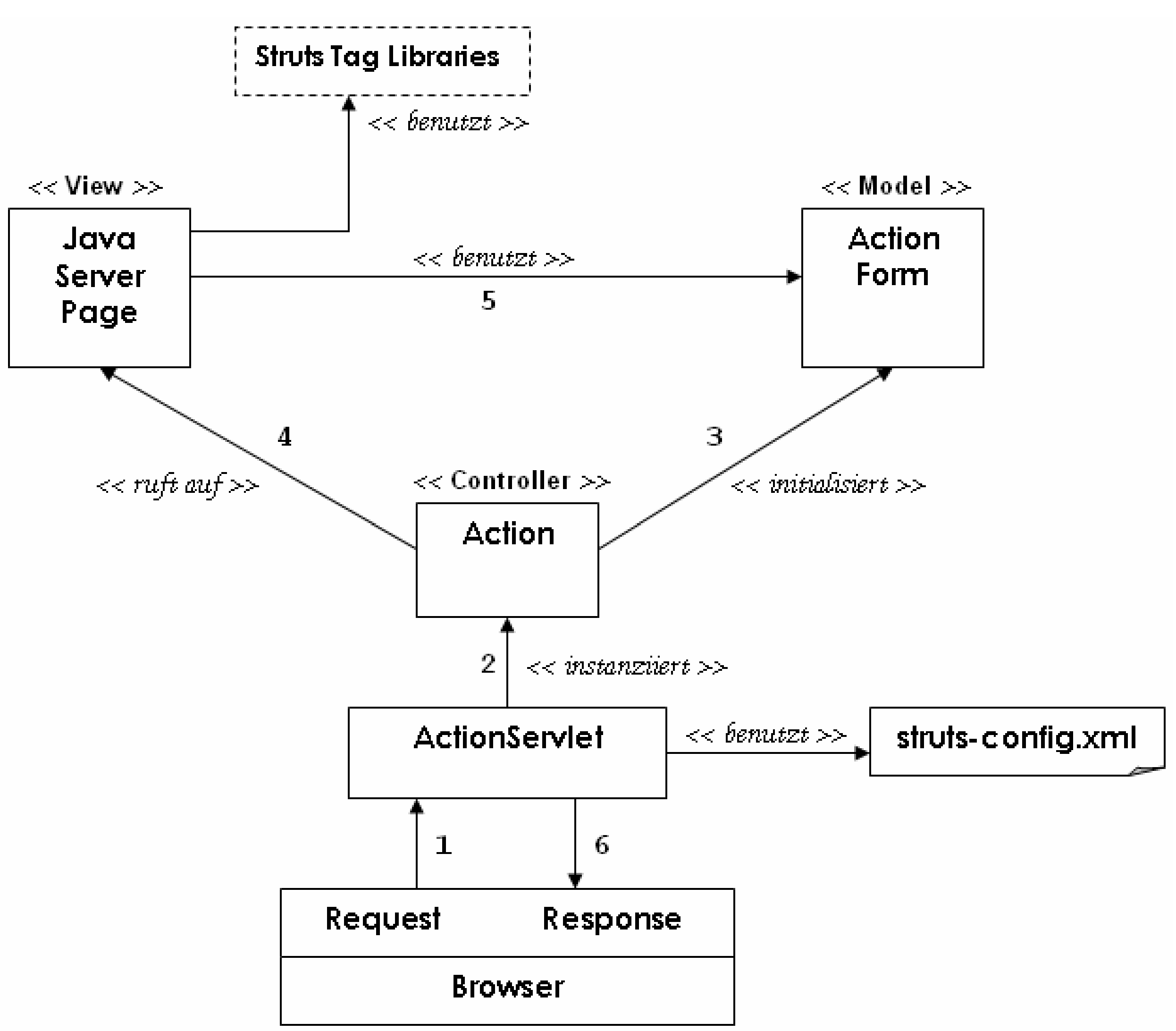


Abbildung 1: Umsetzung des MVC Modells in Struts.
Für Details siehe die folgenden Ausführungen

Die Modelklassen sind reine Datenelemente, deren Inhalte entweder dargestellt oder weiterverarbeitet werden. Das Bindeglied zwischen Modelklassen und Darstellungskomponenten, den Views, sind die Controllerklassen, die die richtige Modelklasse initialisieren, oder falls schon vorhanden, finden und weiterleiten. Es muss also für Controllerkomponenten definiert werden, welche Modelklassen mit welchen Daten zu erzeugen sind, oder aber wo bereits erstellte Datenelementkomponenten gefunden werden können. Außerdem müssen

der Controllerkomponente View Elemente zugewiesen werden, an die sie die Modelklassen zur Darstellung ihrer Informationen weiterleiten können.

Die View Elemente stellen die Informationen der Modelklassen in beliebiger Weise dar. Sie sind zugleich ein möglicher Ausgangspunkt für Datenänderungen. Diese Änderungen werden wieder an den zuständigen Controller weitergeleitet, der die Daten, also die Modelklassen auf den aktuellen Stand bringt, bzw. hält.

Die Implementierung des Zusammenspiels dieser drei MVC Komponenten in Struts aufzuzeigen ist das Hauptthema dieses Kapitels und wird in den Abschnitten 2.3 und 2.4 dargestellt.

- Das Bearbeiten und Weiterleiten von Benutzeraktionen am Browser und die Transformation in ein einheitliches Modell. Dieses frameworkspezifische Modell ermöglicht ein Event -, bzw. Action Handling wie es aus der Entwicklung von Swing, AWT oder Windows Benutzeroberflächen bekannt ist.
- Internationalisierung für die Unterstützung der Mehrsprachigkeit von Webanwendungen.
- Die Möglichkeit, die Applikationslogik aus den View Elementen in die Controller, also Actionklassen auszulagern, um die notwendige Trennung zwischen Daten und Darstellung zu vollziehen. Dies wird durch TAG LIBRARIES ermöglicht, mit denen die Views die Daten aus den Modelobjekten beziehen, die der Controller verwaltet.

In der Literatur, wie auch in den Büchern [1] und [2] wird auf diese Sachverhalte detailliert eingegangen, mit der Absicht, dass der Leser anhand von Beispielen relativ schnell ein erstes Implementierungsergebnis erzielt. Mit diesem Ziel bleiben dem Struts Neuling sinnvollerweise die komplexen Mechanismen, die in diesem Framework stecken, verborgen. Ist er gewillt, mit gewachsener praktischer Erfahrung, mehr über Struts zu erfahren, findet er in den Abschnitten 1.3 und 1.4 detaillierte Informationen über die internen komplexen Abläufe, die Struts zu dem machen, wovon der Webapplikationsentwickler profitiert.

Wann immer auf Quelltext Bezug genommen wird, wird dieser mit der Schriftart `Courier New` mit der Schriftgröße zehn gekennzeichnet. Dies gilt ebenso für alle anderen Kapitel.

Um dem Leser zunächst eine Wissensgrundlage zu geben werden im folgenden Abschnitt grundlegende Elemente von Struts vorgestellt, auf denen die Kapitel ab 1.3 aufbauen.

1.2 Grundstrukturen und Basiskomponenten

Mittlerweile existiert eine große Menge von Literatur wie Bücher und Dokumentations- und Code- bzw. Beispieltutorials. Sie alle ermöglichen bei entsprechenden Programmiervorkenntnissen einen angenehmen Einstieg in das Thema und es lässt sich danach schnell eine eigene mehr oder weniger umfangreiche Webanwendung erstellen. Was jedoch fehlt ist ein tiefer implementierungstechnischer Einblick, welche Klassen an welchen Aktionen beteiligt sind und wie sie zusammenarbeiten, bzw. wie die Aufgabenverteilung der einzelnen Bausteine aus Architektursicht realisiert ist.

Um diese Lücke zu schließen, werden zunächst die wesentlichen Struts Klassen aufgeführt. Es wird je Klasse eine kurze Funktionsbeschreibung gegeben und deren Rolle im Kontext zu den Ausführungen aus Abschnitt 2.3 und 2.4 dargestellt. Auf diese Weise erhält der Leser eine Übersicht über die Bausteine von Struts, die an der implementierungstechnischen Realisierung beteiligt sind. Diese Einführung ist die Vorraussetzung, um die folgende Analyse der Implementierung zu verstehen und nachzuvollziehen.

Die detaillierte Analyse ist als eine Art Reiseführer durch den Struts Quelltext zu verstehen. Daher wird dem Leser empfohlen, sich diesen bei der Jakarta Struts Website runter zu laden, oder von der Diplomarbeits CD zu kopieren und sich nebenher die Klassen anzusehen, die die Analyse gerade bahandelt.

Im Grobüberblick besteht Struts aus:

- Core JAR BIBLIOTHEKEN, die die Klassen aus der folgenden Tabelle enthalten
- TAG LIBRARIES ebenfalls in Form von JAR BIBLIOTHEKEN und `.tld` Dateien, die für das Design von JSP Seiten verwendet werden
- JAR COMMONS BIBLIOTHEKEN `(commons-xyz.jar)` die externe Services und Werkzeuge enthalten für z.B. xml Parsing, dynamische Objekterzeugung und Methodenaufrufe, Requestbearbeitung, Validierung, Logging und Dateiupload u.v.a.. Von besonderer Bedeutung ist `commons-digester.jar` für das Parsen der `struts-config.xml`.

Die wichtigen Bausteine und Klassen von Struts sind:

Paket **org.apache.struts.action**
`org.apache.struts.action.Action`
Repräsentiert die Controller – Komponente innerhalb von MVC. Jede Instanz dieser Klasse implementiert eine Aktionsausführungsmethode, in der die beliebig komplexe Applikations-logik zu dem auszuführenden Request steht
`org.apache.struts.action.ActionForward`
Ein Navigationszielobjekt, wohin, d.h. auf welche Seite oder nächsten Zwischenschritt der Bearbeitung ein Request geleitet wird, nachdem ein Benutzer einen Link oder einen Submit Button angeklickt hat. Es repräsentiert ein `<action>` Subelement `<forward>` in einer `struts-config.xml`.
`org.apache.struts.action.ActionMapping`
Es stellt alle Daten zur Verfügung, die in `der struts-config.xml` unter dem `path` Attribut eines `action` Elements hinterlegt sind. Mittels dieses Objekts weiß das Framework, welche Actionklasse es per Reflection erzeugen muss und welche Modelkomponente diesem Controller zur Verfügung gestellt werden muss. Das `ActionMapping` steht bei jeder `Action` Instanz zur Weiterleitung und Bearbeitung eines Requests bereit, damit dir richtigen Daten auf der richtigen JSP – View zur Darstellung gebracht werden.
`org.apache.struts.action.ActionForm`
Repräsentiert die Model – Komponente innerhalb von MVC. Eine Ableitung dieser Klasse ist standardmäßig eine reine Datenattributklasse mit `get` und `set` Zugriffsmethode je Attribut
`struts-config.xml`
Die `struts-config.xml` kann als eine Form der `Registry` von Windows verstanden werden. So wie in der Registry Pfade und Informationen zu allen installierten Programmen gespeichert, sowie Komponenten wie OLE – Objekte registriert sind, wird in der `struts-config.xml` alles hinterlegt, was für die Webapplikation von Bedeutung ist. Dies sind alle MVC Komponenten und deren Zusammenwirken beim Ablauf der Anwendung. Konkret heisst das, dass die xml Datei aus `action` Entitäten, bzw. Tags besteht in der der voll klassifizierende Klassenname, ein Verweis auf die dazugehörende Model-komponente, also `ActionForm` – Instanz, sowie die Viewelemente stehen, auf die abhängig von der Applikationslogik verzweigt wird. Die Modelkomponenten werden als `form-bean` Entitäten, bzw. Tags notiert. Auch hier steht der voll klassifizierende Klassenname des Models und ein Referenzattribut, das die `action` Entität benutzt, um auf das Model zu verweisen.

org.apache.struts.action.ActionServlet

Das ActionServlet ist das Herz einer jeden Struts Webapplikation. Es ist dafür zuständig, dass die Bearbeitung jeder vom Benutzer eingeleieteten Interaktion angestoßen wird. D.h., dass ein eingegehender Request zuerst hier landet, analysiert wird und danach an das richtige Bearbeitungsobjekt gegeben wird, das die richtigen Ergebnisse liefert, nachdem jegliche Art notwendiger Applikationslogik abgearbeitet wurde, um die notwendigen Daten zu beschaffen.

Um die interne Funktionsweise braucht sich der Entwickler nicht zu kümmern. Struts ist so organisiert, dass er sich auf die reine Anwendungsentwicklung konzentrieren kann. Es ist lediglich notwendig, dass im Deployment Deskriptor `web.xml` eingetragen wird, welche Klasse die Rolle des `ActionServlet` übernimmt. Im einfachsten Fall wird es selbst verwendet, oder eine Klasse, die von `ActionServlet` erbt. Bevor die Webanwendung gestartet wird, bearbeitet es die Daten, aus der `struts-config.xml`. Es speichert alle MVC Komponenten, bestehend aus den Informationen aus den `form-bean` und `action` Elementen.

Damit das `ActionServlet` seine Aufgaben erfüllen kann und die interne Funktionsweise transparent bleibt, stehen ihm eine Reihe von Hilfsklassen zur Verfügung. Die wichtigste ist der `RequestProcessor` für die Requestweiterleitung und Konfigurationsdatenklassen für Einträge aus `struts-config.xml` und `web.xml`.

org.apache.struts.action.RequestProcessor

Er beinhaltet das Request processing und wird vom `ActionServlet` initialisiert. Objekte, die bearbeitet werden, also entweder beschafft oder ausgewertet werden müssen, sind folgende:

```
java.util.Locale
org.apache.struts.action.Action
org.apache.struts.action.ActionForm
org.apache.struts.action.ActionForward
org.apache.struts.config.ForwardConfig
org.apache.struts.action.ActionMapping
```

Die requestspezifischen Instanzen dieser Klassen stehen in einem Registryobjekt des Typs `org.apache.struts.config.ModuleConfig`, das vom `ActionServlet` initialisiert wurde. Per Schlüsselattribut, das jedem Request zur Verfügung steht, wird dem `RequestProcessor` mitgeteilt, welche der Instanzen für die Requestbearbeitung notwendig sind.

Die folgenden Klassen werden von den beiden Haupt – Struts – Klassen `ActionServlet` und `RequestProcessor` genutzt. Mit ihnen hat der Entwickler beim Entwicklungsprozess nichts zu tun, da deren Verwendung vollkommen gekapselt und transparent ist. Seit der Struts Version 1.1 wurde im Zuge der Modularisierung von Struts Webanwendungen mehr Funktionalität vom `ActionServlet` in den `RequestProcessor` ausgelagert.

Paket **org.apache.struts.config**

org.apache.struts.config.ActionConfig

Repräsentiert ein `action` – Element samt Unterelementen aus der `struts-config.xml`. Folgende Informationen sind durch diese Klasse verfügbar:

`path`: Der Name unter dem die Aktion ausgelöst wird
`type`: Der voll klassifizierende Klassenname der Controller – Action Komponente
`scope`: Die Sichtbarkeit der über die `ActionForm` verschickten Daten. Entweder nur für einen Request oder für alle Requests dieser Session und dieses Benutzers.
`forward`: Eine Zielseite oder wiederum andere Aktionen auf die nach der Abarbeitung der Controller Action Komponente verzweigt wird.

org.apache.struts.config.ControllerConfig

Repräsentiert ein `controller` – Element aus der `struts-config.xml`. Seit Version 1.1 können verschiedene Module einer Struts Webapplikation ihre eigenen Konfigurationsparameter

setzen.

Jedes Modul kann optional seine eigene Implementierung von `org.apache.struts.action.RequestProcessor` hinterlegen, wenn eine spezielle `http` Requestbehandlung erforderlich ist. Ferner kann hier der Umgang mit Dateiuploads geregelt werden. Z.B. die maximale Größe einer Datei.

`org.apache.struts.config.DataSourceConfig`

Repräsentiert ein `data-source` – Element aus der `struts-config.xml`. Der Zweck dieses Elements ist die Vereinfachung der Bereitstellung von Datenbank JDBC Ressourcen, wie z.B. `java.sql.Connection`. Darüber hinaus wird das Struts Framework in das Management des Lebenszyklus von `Connection`s eingebunden. Ein einheitlicher Zugriff auf diese Ressourcen führt zu einer Reduzierung der Fehlerwahrscheinlichkeit.

Dieser einheitliche Zugriff wird über das Fabrikmuster realisiert. Erreichbar sind die Datenbankressourcen über eine von `javax.sql.DataSource` abgeleitete Fabrikklasse, deren Eigenschaften (z.B. DB – Treiberklassen oder DB - URL) unter dem `data-source` Element spezifiziert ist.

Zugegriffen wird auf diese Eigenschaften über `org.apache.struts.config.DataSourceConfig`, die die `javax.sql.DataSource` Implementierung `org.apache.struts.util.GenericDataSource` bereitstellt.

`org.apache.struts.config.FormBeanConfig`

Repräsentiert ein `form-bean` – Element, also eine Modelklassendeklaration aus der `struts-config.xml`. Folgende Attribute sind zu setzen, um das Model über das Struts Framework zu verwenden:

`name`: Der Name unter dem das Model referenziert, bzw. erreichbar ist
`type`: Der voll klassifizierende Klassenname der Model Komponente

Seit Struts 1.1 gibt es ein weiteres optionales Attribut zur Vereinfachung der Handhabung und Entwicklung von Model Klassen.

`dynamic`: `true` – Das Model erbt von `DynaActionForm`
`false` – Das Model ist selbst `DynaActionForm` oder erbt von `ActionForm`
`className`: Kann weggelassen werden, wenn keine eigene Implementierung von `org.apache.struts.config.FormBeanConfig` existiert.

`org.apache.struts.config.ForwardConfig`

Repräsentiert ein `forward` – Unterelement aus der `struts-config.xml`.

`name`: Der Name unter dem der Wert unter `path` aufgerufen wird und damit verfügbar ist
`path`: Die jsp Seite oder der Name der nächsten Aktion auf die als nächstes verzweigt wird.

`org.apache.struts.config.MessageResourcesConfig`

Repräsentiert ein `message-resources` – Element aus der `struts-config.xml`.

`parameter`: Der vollklassifizierende Pfad (analog zu einem Klassennamen) zu einer `.properties` Datei. Sie enthält unter Wert – Schlüssel – Paaren alle Labels, die auf einer generierten HTML Seite verwendet werden. Dieser Mechanismus unterstüzt auch die Internationalisierung. Mit dem Prefix: _<locale Abkürzung>, wie z.B. `_de` oder `_fr` zu `com.test.example.ApplicationResources_fr.properties` kann die gewünschte Sprache der HTML Seite verwendet werden.

`key`: Optional. Ein Zugriffsschlüsselattribut, unter dem die Datei intern hinterlegt ist und besser gefunden werden kann.

`org.apache.struts.config.ModuleConfig`

Seit Struts 1.1 lässt sich eine Webapplikation in Subapplikationen aufteilen. Dies macht dann Sinn, wenn aufgrund eines großen Umfangs einer Anwendung ohne Modularisierung die Übersichtlichkeit verloren geht. Jedem Modul ist eine eigene `struts-config.xml` zugeordnet.

Dabei ist ein `ModuleConfig` Objekt eine Javaobjektrepräsentation, das alle in der `struts-config.xml` enthaltenen Informationen bereitstellt. Dazu zählen u.a. die Informationen in und unter den Tags `action`, `form-bean`, `forward`. Diese werden für die Beschaffung und die Bearbeitung von darzustellenden Daten durch die Controller `Action` - Klasse benötigt.

org.apache.struts.config.PlugInConfig

Repräsentiert ein `plug-in` – Element aus der `struts-config.xml`. Es gehört atomar zu einem Modul innerhalb einer Struts 1.1 Applikation. Daraus folgt, dass einem `PlugIn`, das das Interface `org.apache.struts.action.PlugIn` implementieren muss, immer ein `ModuleConfig` Objekt mitgegeben werden muss. Ein `PlugIn` dient zur Initialisierung aller Arten von Ressourcen wie z.B. von Datenbank Verbindungen, oder Validierungsregeln für Benutzer-eingaben, die für das gesamte Modul von Bedeutung sind und somit nicht für jeden Zugriff neu ermittelt oder erzeugt werden müssen.

Das Framework initialisiert das `PlugIn`, wenn in der `struts-config.xml` folgende Attribute und Unterelemente vorhanden sind:

`className`: Der vollklassifizierende Klassenname der PlugIn - Klasse
`id`: Der Name unter dem auf das `PlugIn` referenziert werden kann (optional)

Mittels des optionalen Unterelements `set-property` kann mit dem Wert des Attributs `property` auf den Wert des Attributs `value` zugegriffen werden.

org.apache.struts.config.ModuleConfigFactory

Erzeugt ein `ModuleConfig` Objekt gemäß des request Prefixes, das in der `web.xml` des Hauptmodules unter einem `init-param` Element spezifiziert wurde. Z.B.:

```
<init-param>
  <param-name>config/examples</param-name>
  <param-value>/WEB-INF/struts-examples-config.xml</param-value>
</init-param>
```

Dem erzeugten `ModuleConfig` Objekt wird im Konstruktur der Wert `config/examples` übergeben. So ist bekannt, aus welcher `struts-config.xml`, nämlich `struts-examples-config.xml`, das `ModuleConfig` Objekt in den entsprechenden Methoden mit Daten gefüllt werden muss.

Paket **org.apache.struts.util**

org.apache.struts.util.GenericDataSource

Ist auf deprecated gesetzt. Sie wurde durch `org.apache.commons.dbcp.BasicDataSource` ersetzt. Diese Klasse stammt aus dem `commons-dbcp` Paket von Jakarta, einem Datenbankframework.

org.apache.struts.util.MessageResources

Abstrakte Basisklasse, die die Mehrsprachigkeit einer Webapplikation ermöglicht. Eine Defaultimplementierung ist die Klasse `org.apache.struts.util.PropertyMessageResources`, die Struts bei der Initialisierung der Anwendung benutzt. In `.properties` Dateien werden Labels, die auf einer JSP Seite stehen sollen, unter Schlüsseln hinterlegt. Konvention ist die folgende Notation: `key.to.label=Anzuzeigender Text`

Um Mehrsprachigkeit zu erhalten muss lediglich die gewünschte Sprachversion der `.properties` Datei geschrieben werden. Lautet der Name dieser Datei z.B. `ApplicationResources.properties` heißen die anderssprachigen `ApplicationResources_`**`xy`**`.properties`, wobei `xy` für die jeweilige zweibuchstabige Länderkennung steht, also für deutsch `de`, französisch `fr`. Diese Endung wird als `Locale` bezeichnet. Die `ApplicationResources. properties` ist hierbei die default `Locale`.

In jeder dieser Dateien sind die Schlüsselnamen jeweils dieselben. Der Wert ist der Label in der übersetzten Sprache. Die englische und französische Version sieht folgendermaßen aus:

```
ApplicationResources_en.properties: key.to.label=Label to display
ApplicationResources_fr.properties: key.to.label=text à montrer
ApplicationResources.properties:    key.to.label=Anzuzeigender Text
```

Mit diesem Texthandling auf JSP Seiten sollen hart kodierte Inhalte vermieden und die Übersichtlichkeit verbessert werden, sodass bei notwendigen Textänderungen nicht in die eigentliche Anwendung eingegriffen werden muss.

Damit dieser gesamte Mechanismus genutzt werden kann, muss der Entwickler folgende Konfigurationseinstellungen vornehmen:

1. in der `web.xml` muss folgender Eintrag stehen:

```
<web-app>
  <servlet>
    <servlet-name>action</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
    <init-param>
      <param-name>application</param-name>
      <param-value>ApplicationResources</param-value>
    </init-param>
   …
  </servlet>
…
</web-app>
```

Der Parametername `application` ist reserviert in Struts und beim Start einer Webanwendung wird über diesen Schlüssel der Parameterwert, hier `ApplicationResources`, abgefragt. Dieser Wert muss mit der Datei **`ApplicationResources`**`.properties` übereinstimmen, damit der Struts Framework die Inhalte aller `ApplicationResources_xy.properties` Dateien laden und zur Laufzeit der Anwendung zur Verfügung stellen kann.

2. Alternativ und seit Struts 1.1 üblich sind die Informationen diesbezüglich in die `struts-config.xml` überführt worden. Dies hat zwei Vorteile:
   a. Besteht eine Webanwendung aus mehreren Teilanwendungen, kann jede ihr eigenes `MessageResource` Paket definieren und verwenden.
   b. Es können mehrere `MessageResource` Bundles verwendet werden.

   Notiert werden die `MessageResource`s folgendermaßen (siehe auch Abschnitt unter `org.apache.struts.util.MessageResourcesConfig`):

```
<struts-config>
   …
  <message-resources parameter="com.test.one.ApplicationResources"
                     key="alternate"/>
  <message-resources parameter="com.test.two.AlternativeResources"/>
   …
</struts-config>
```

3. Die Dateien `.properties` müssen über die `classpath` Variable der Webapplikation verfügbar sein. Gewöhnlich sind sie im `WEB-INF/classes` Verzeichnis der Webanwendung zu finden.

`org.apache.struts.util.MessageResourcesFactory`

Jede `MessageResourcesFactory` – Fabrikklasse muss von dieser abstrakten Klasse erben und die Methode `createResources()` implementieren. Diese erzeugt ein Objekt, das von `org.apache.struts.util.MessageResources` erbt. Struts liefert Basisimplementierungen von `MessageResources` und `MessageResourcesFactory`. Diese werden verwendet, wenn in der `struts-config.xml` bei den Tags `message-resources` keine Eigenimplementierung unter deren Attributen `className` und `factory` deklariert ist, sprich wenn sie weggelassen werden.

org.apache.struts.util.RequestUtils

Eine Helferklasse für das Verarbeiten von Requests. Zu den Services dieser Klasse gehören:

Service	Nutzer
Das Suchen nach Labeln in `MessageResources.properties` Dateien. Methode: `String` **`message`**`(PageContext pageContext,String bundle,String locale,String key,Object args[])`	`MessageTag.doStartTag()`

Das dynamische Laden von Klassen zur Laufzeit. Methode:
`RequestUtils.`**`applicationInstance`**`(String clazz)`
	`ActionServlet`
	`MessageResourcesFactory`
	`RequestProcessor`
	`ModuleConfigFactory`
	`MessageResourcesFactory`
	`RequestUtils`

URL Bearbeitung und Formatierung. Methoden: • `URL` **`absoluteURL`**`(HttpServletRequest request, String path)` • `String` **`getActionMappingURL`**`(String action, PageContext pageContext)` • `String` **`printableURL`**`(URL url)` • `String` **`actionURL`**`(HttpServletRequest request,ActionConfig action,String pattern)` • `String` **`forwardURL`**`(HttpServletRequest request, ForwardConfig forward)` • `String` **`pageURL`**`(HttpServletRequest request, String page)` • `URL` **`requestURL`**`(HttpServletRequest request)` • `URL` **`serverURL`**`(HttpServletRequest request)`	`RequestProcessor` `RequestUtils`
Die Zuordnung eines Applikationsmoduls zu einem Request. Methode: `void` **`selectModule`**`(String prefix, HttpServlet Request request, ServletContext context)`	`ActionServlet.process()`
Das Ermitteln von ActionForm Instanzen zu einem Request. Methode: `ActionForm` **`createActionForm`**`(HttpServletRequest request, ActionMapping mapping, ModuleConfig moduleConfig, ActionServlet servlet)`	`RequestProcessor`
Das Ermitteln von Bean Attributen im angegeben Scope. Ein Bean kann ein beliebiges Objekt sein das request-, oder session weit gespeichert wurde `Object` **`lookup`**`(PageContext pageContext, String name, String property, String scope)`	`BeanUtils` `MethodUtils`
Schreibt Attribute unter Verwendung von Reflection Mechanismen in die unter Parameter `bean` angegebene Beanklasse. Die Attribute können z.B. Daten aus HTML Form Elementen sein, wie Textfelder. Hierbei kommt auch die Utilityklasse `BeanUtils` zum Einsatz Methode: `void` **`populate`**`(Object bean,String prefix,String suffix,HttpServletRequest request)`	`RequestProcessor` `SetPropertiesRule`

Paket **org.apache.struts.digester**
org.apache.commons.digester.Digester
Ein Digester dient dazu xml Dateien zu parsen und daraus Java Objekte zu erzeugen. Dazu müssen Regeln (org.apache.commons.digester.Rule) oder Regelsätze (org.apache.commons.digester.RuleSet) definiert und der Digester damit konfiguriert werden. So werden z.B. die Daten aus der web.xml und aus struts-config.xml Dateien extrahiert und in Objekten verfügbar gemacht, die der Webserver, bzw. das Framework benutzt. Ursprünglich stand der Digester nur innerhalb für Struts und den Tomcat Webserver zur Verfügung, bis er in das Archiv commons-digester.jar ausgelagert wurde. Implementierungstechnisch ist der Digester eine xml Parserklasse, da er als Erweiterung von org.xml.sax.helpers.DefaultHandler alle Methoden implementiert, die von den Interfaces ContentHandler, DTDHandler, EntityResolver und ErrorHandler kommen. Er ist somit ein Ereignis getriebener SAX Parser, da er keinen DOM Elementbaum speichert, sondern Regeln (org.apache.commons.digester.Rule), die ihm für ein xml Element hinzugefügt wurden, ausführt und beendet. Ersichtlich wird dieses Verhalten bei den callback Methoden startElement und endElement des Interfaces ContentHandler. Der RequestProcessor benutzt den Digester für die modul-, oder applikationsspezifische struts-config.xml, um dessen Objektrepräsentation ModuleConfig zu erzeugen. Das Action Servlet verwendet ihn zur Speicherung der Inhalte der web.xml.
org.apache.commons.digester.Rule
Eine Rule ist eine Implementierung eines spezifischen Verhaltens für ein gegebenes xml Element. Es gibt eine Reihe vorgefertigter Rules u.a. zum Erzeugen von Wrappern und das Setzen ihrer Attribute für struts-config.xml Elemente, z.B. FormBeanConfig für das Element <form-bean> Das Erzeugen von Java Objekten übernimmt die ObjectCreateRule. Für das Setzen der Attribute stehen die SetPropertyRule für ein und die SetPropertiesRule für das Setzen mehrerer Attribute zur Verfügung.
org.apache.commons.digester.RuleSet
Ein RuleSet ermöglicht das Laden mehrerer Rules in einen Digester. Das Interface deklariert eine Methode addRuleInstances, die eine Implementierungsklasse wie z.B. org.apache.struts.config.ConfigRuleSet speziell für eine struts-config.xml bereitstellen muss.

Da der Leser nun einen Überblick über die zentralen Komponenten und deren Hilfsbausteine gewonnen hat, folgt im nächsten Schritt eine detaillierte Analyse des Ablaufs und des Zusammenspiels der Klassen, aus denen das Struts Framework besteht. Dieses Zusammenspiel wird mit zwei Szenarien betrachtet: Der Ablauf des Hochfahrens, d.h. vor allem die Initialisierung aller benötigten Resourcen einer Webapplikation bis zur Anzeige der Startseite, Kapitel 2.3. Kapitel 2.4 untersucht die Struts Internas, die für die Bearbeitung von User Requests sorgen.

1.3 Die Initialisierung einer Struts Webapplikation

Die Initialisierung beginnt direkt nach dem Hochfahren des Tomcat Webservers. Dieser startet am Ende seiner Initialisierungsphase alle Webapplikationen automatisch, die im Verzeichnis `%TOMCAT_HOME%\webapps` zu finden sind. Ebenfalls berücksichtigt werden spezielle Einträge in der zentralen Konfigurationsdatei `server.xml` unter `%TOMCAT_HOME%\conf`. Diese zeigen auf Webapplikationshauptverzeichnisse, die nicht im Verzeichnis `%TOMCAT_HOME%\webapps` liegen. Sie befinden sich unter dem Element `/Server/Service/Host/` und haben folgendes Format:

```
<Context path="/optional"
         docBase="web.app.main.dir"
         workDir=" web.app.main.dir/work/org/apache/jsp"
         debug="0" privileged="true"/>
```

Die letzte Methode, die Tomcat beim Starten ausführt, ist wie im folgenden Listing ersichtlich `ApplicationFilterChain.internalDoFilter()`. Von hier aus scannt Tomcat alle `web.xml` Deployment Deskriptoren, die er in jeder registrierten Webapplikation findet. In jedem Deployment Deskriptor startet er das Init Servlet, dessen voll klassifizierenden Klassennamen er unter dem Element `web-app/servlet/servlet-class` findet. Bei einer Strutsapplikation ist dies entweder direkt das `ActionServlet` oder eine davon vom Applikationsprogrammierer abgeleitete Klasse. Anhang A zeigt den genauen Ablauf, was nun als nächstes passiert im Überblick.

Bei einer Struts Webanwendung beginnt alles mit `ActionServlet.init()`. Von diesem Startpunkt aus werden alle Ressourcen und Konfigurationsdateien, wie `web.xml`, alle `struts-config.xml` Dateien, sowie `.properties` Dateien, die die Labels auf den JSP Seiten und deren Mehrsprachigkeit enthalten ausgewertet, um dem Entwickler die Werkzeuge zur Verfügung zu stellen, die bei der Erstellung der Anwendung braucht.

Schritt 1: `ActionServlet.initInternal()`

Die erste Methode innerhalb von `ActionServlet.init()` ist `ActionServlet.initInternal()`. In ihr wird ein `MessageResources` Objekt erzeugt. Dies geschieht mit der statischen Methode `MessageResources.getMessageResources("org.apache.struts.action.ActionResources")`. Der Übergabeparameter bestimmt den Namen der `.properties` Datei, die alle Struts Standardfehlermeldungen enthalten, die dem Entwickler angezeigt werden, wenn er eine Konvention aufgrund eines Fehlers nicht eingehalten hat. Z.B. wenn ein einer von `ActionForm` abgeleiteten Klasse das `extends ActionForm` fehlt, erscheint die Fehlermeldung, bzw. Exception:

```
Exception creating bean of class com.tsystems.struts.user.struts.UserLoginForm: {1}
```

In `MessageResources.getMessageResources()` Methode wird zunächst eine Fabrikklasse erzeugt. Die Methode `MessageResourcesFactory.createFactory()` dieser abstrakten Fabrikklasse erzeugt die von Struts mitgelieferte Defaultfabrikimplementierungsklasse `PropertyMessageResourcesFactory`. Dazu benutzt `MessageResourcesFactory.createFactory()` die Methode `RequestUtils.applicationClass(factoryClassName)`, die per `ClassLoader` die `PropertyMessageResourcesFactory` in den Speicher lädt. Mittels dieser Fabrik erzeugt `MessageResources` mit der Methode `PropertyMessageResourcesFactory.createResources()` die Standardimplementierung `PropertyMessageResources`.

Diese Klasse implementiert die entscheidende Methode `getMessage(Locale, String message)` der abstrakten Basisklasse `MessageResources`, um auf die Inhalte aller `MessageResources[_xy].properties` Dateien zugreifen zu können. `MessageResources` stellt eine Reihe

von Servicemethoden, bzw. Varianten für getMessage() bereit. Diese haben jeweils unterschiedliche Parameter und dienen dazu, die Methode PropertyMessageResources.getMessage(Locale, String message) bequemer und effektiver nutzen zu können. So wird z.B. mit Message Resources.getMessage(Locale locale, String key, Object args[]) die Parametrisierung von Labels unterstützt: In einer MessageResources.properties Datei kann es Labels unter Schlüsseln geben wie label.name=label {0} must be displayed under {1}. Diese holt sich die Klasse MessageResources mittels der Methode PropertyMessageResources.getMessage(Locale, String message). MessageResources formt dann den Wert unter dem Schlüssel label.name so um, dass die Werte von Object args[] in die Platzhalter eingesetzt werden. Dies geschieht über den Aufruf der Methode java.text.MessageFormat.format(Object args[]).

Als nächstes stellt sich die Frage, wie die Schlüssel und Werte aller verschiedenen .properties Dateien, d.h. die Labels und deren Schlüssel aller Sprachvarianten verwaltet, bzw. gespeichert werden. Hier für gibt es die Methode PropertyMessageResources.loadLocale(localeKey). localeKey ist immer die zweibuchstabige Länderkennung, also de, fr, en, usw. In dieser Methode passiert folgendes:

Zunächst wird untersucht ob die Locale, also localeKey bereits vorhanden ist. Falls nein wird er in der HashMap locales gespeichert. Um alle .properties Dateien zu erfassen wird auf das String Attribut config der Basisklasse MessageResources zugegriffen. Dieses enthält den voll klassifizierenden Package Namen mit Dateinamen. Dieser wurde vom ActionServlet definiert. Hier ist dies der Wert org.apache.struts.action.ActionResources. In einer Webanwendung wird dieser Wert durch die Deklaration innerhalb eines <message-resources> Elememts in einer struts-config.xml oder einem Äquivalent einer Subapplikation deklariert und mittels der Klassenrepräsentation MessageResourcesConfig dem Struts Framework zugänglich gemacht.

Im letzteren Fall besteht im Gegensatz zum ersten die Möglichkeit, dass es mehrere Sprachversionen gibt. Daher wird der Übergabeparameter localeKey der Methode nach einem Underscore _ an den Package Namen angehängt. Das Ergebnis: org.apache.struts.action.ActionResources_[localKey]. Da es sich aber nicht um einen Klassenamen handelt, muss noch die Extension .properties hinzugefügt werden. Je nach Locale, bzw. Wert von localeKey können so mit der Methode PropertyMessageResources.loadLocale(String localeKey) alle Sprachvarianten gefunden werden.

Wird z.B. in einer struts-config.xml unter <struts-config>

```
<message-resources parameter="com.test.one.ApplicationResources"/>
```

oder in der web.xml folgendes definiert:

```
<init-param>
  <param-name>application</param-name>
  <param-value>ApplicationResources</param-value>
</init-param>
```

findet die Methode sofern angelegt, alle Sprachvarianten je nach übergebenem Parameter localeKey. Also:

ApplicationResources_en.properties

ApplicationResources_fr.properties, usw.

Wie üblich werden die Inhalte von .properties Dateien in java.util.Properties Objekten gespeichert. Dazu wird über einen ClassLoader dessen Methode getResourceAsStream(String fileName) ein InputStream erzeugt und mittels java.util.Properties.load(InputStream) geladen.

Da es mehrere Sprachvariaten dieser `.properties` Dateien gibt, ist es naheliegend, für jede ein solches `java.util.Properties` Objekt anzulegen. Dies regelt `PropertyMessageResources` aber anders. Es wird nur eine `java.util.HashMap` über alle Schlüssel und deren Labels aller `.properties`, Dateivarianten bzw. `java.util.Properties` Objekten geführt. Dazu ist es nötig, den Schlüsseln ihre zugehörige Locale anzuhängen und darunter dann den jeweiligen Wert aus den `Properties` zu speichern. Dies wird in der `loadLocale` Methode so realisiert:

```
String extKey = messageKey(localeKey, key);
java.util.HashMap.put(extKey, props.getProperty(key))
```

Die kleine Hilfsmethode `messageKey()` hängt jedem Schlüssel aus `java.util.Properties` einen `"."` und den `localeKey` an. Das Ergebnis ist dann der neue Schlüssel des Labels. In der `HashMap` liegt ein Label in allen Übersetzungen so vor:

```
label.name.de=label {0} muss unter {1} angezeigt werden
label.name.fr=label {0} doit être montrer sous {1}
label.name.en=label {0} must be displayed under {1}
```

Da nun die Methode `ActionServlet.initInternal()` abgearbeitet wurde, stellt sich die Frage, wie die Inhalte der `.properties` Dateien vom Entwickler abgerufen werden. Der Entwickler benutzt dazu die Struts Tag Libraries. Hier genauer gesagt den Tag `org.apache.struts.taglib. bean.MessageTag`. Dessen Definition befindet sich in der `tld` Datei `struts-bean.tld`. Diese besagt, dass zu dem `message` Tag die folgenden optionalen Attribute gesetzt werden können:

Attribut	Funktion

`arg0 arg1 arg2 arg3 arg4`	Für optionale Werte. Diese werden für die Platzhalter {n} in den Werten der Labels aus den `.properties` Dateien eingesetzt. Z.B. Für
	`arg0 = Das Objekt`
	`arg1 = dem Hinweis`
	wird aus: `{0} muss unter {1} angezeigt werden`
	`Das Objekt muss unter dem Hinweis angezeigt werden`

`bundle`	Welche Locale, d.h welche Sprache verwendet werden soll
`key`	Der Wert des gewünschten Labels, der angezeigt werden soll
`locale`	Falls definitiv eine bestimmte Sprachversion des Werts eines Labels
`name`	Der Name des `ActionForm` Beans wenn keine `.properties` Datei verwendet werden soll
`property`	Das Attribut des `ActionForm` Beans wenn keine `.properties` Datei verwendet werden soll
`scope`	Der Scope in dem nach dem angegebenen `ActionForm` Bean gesucht werden soll

Tabelle 1: Attribute des `<message>` Tag aus den Struts Tag Libraries

Für jedes dieser Attribute besitzt `org.apache.struts.taglib.bean.MessageTag` eine entsprechende `set` Methode, die automatisch aufgerufen wird. Dieser Aufruf funktioniert folgendermaßen: Aus der JSP die u.a.

```
<bean:message key="label.name" arg0="wert0" arg1="wert1"
              arg2="wert2" arg3="wert3"/>
```

enthält, generiert die JSP Engine von Tomcat ein Servlet. An der entsprechenden Stelle der JSP steht im Quelletext des generierten Servlets:

```
…
org.apache.struts.taglib.bean.MessageTag _jspx_th_bean_message_1 =
new org.apache.struts.taglib.bean.MessageTag();

_jspx_th_bean_message_1.setPageContext(pageContext);
_jspx_th_bean_message_1.setParent(null);
_jspx_th_bean_message_1.setKey("label.name");
_jspx_th_bean_message_1.setArg0("wert0");
_jspx_th_bean_message_1.setArg1("wert1");
_jspx_th_bean_message_1.setArg2("wert2");
_jspx_th_bean_message_1.setArg3("wert3");
try {
  int _jspx_eval_bean_message_1 = _jspx_th_bean_message_1.doStartTag();
  if (_jspx_eval_bean_message_1 == javax.servlet.jsp.tagext.BodyTag.EVAL_BODY_BUFFERED)
    throw new JspTagException("Since tag handler class org.apache.struts.taglib.bean.MessageTag
    does not implement BodyTag, it can't return BodyTag.EVAL_BODY_TAG");

  if (_jspx_eval_bean_message_1 != javax.servlet.jsp.tagext.Tag.SKIP_BODY) {
    do {

    } while (_jspx_th_bean_message_1.doAfterBody() == javax.servlet.jsp.tagext.BodyTag.
                                                       EVAL_BODY_AGAIN);
  }

  if (_jspx_th_bean_message_1.doEndTag() == javax.servlet.jsp.tagext.Tag.SKIP_PAGE)
    return;
  } finally {
    _jspx_th_bean_message_1.release();
  }
…
```

Abbildung 2: Servletgenerierung einer JSP mit <message> tag Ausschnitt.

Dieses Beispiel verdeutlicht die Verwendung des `<message>` Tags zum Zugriff auf Labels, die in `MessageResources .properties` Dateien hinterlegt sind.

Nach dem Setzen aller Werte durch Aufruf der entsprechenden `set` Methoden durch das generierte Servlet werden anschließend alle Callback Methoden, wie `MessageTag.doStartTag()`, oder `MessageTag.doEndTag` und `MessageTag.release()` aufgerufen, um in der `MessageTag` Implementierungsklasse mit den gesetzten Attributen, bzw. Informationen den richtigen HTML Code zu erzeugen. Mit anderen Worten: Das `MessageTag` Objekt hat jetzt alle nötigen Informationen, um in seiner vom Servlet aufgerufenen `MessageTag.doStartTag()` Methode mittels des richtigen Labels den zu setzenden Attributen für die Platzhalter, den richtigen Wert in der richtigen Sprache aus den zugehörigen `.properties` Dateien holen zu können und ihn in das generierte HTML einzubetten. In der Methode `MessageTag.doStartTag()` passiert dazu folgendes:

Mittels der Hilfsklasse `RequestUtils` und einer deren statischen Hilfsmethoden `Request Utils.message(PageContext pageContext, String bundle, String locale, String key, Object args[])` wird zunächst nach dem gewünschten `MessageResources` Objekt gesucht. Berücksichtigt wird hierbei der Parameter `bundle`, der ja optional in der JSP bei der Nutzung des `<message>` Tags angegeben wurde. Der erste Parameter des Typs `PageContext` steht für jede JSP zur Verfügung. Dieser wird ebenfalls über das generierte Servlet mittels der abstrakten Fabrik `javax.servlet.jsp.JspFactory` und deren Implementierung `org.apache. jasper.runtime. JspFactoryImpl` über die Methode `getPageContext(Servlet servlet, ServletRequest, ServletResponse,…)` erzeugt. Anschließend wird er mittels der Methode `TagSupport. setPageContext(PageContext)` jeder Tagimplementierungsklasse zur Verfügung gestellt.

Dieser `PageContext` ist die Informationszentrale für alle Daten, die für ein Servlet, bzw. JSP relevant sind. Diese sind mit der entsprechenden `get` Methode verfügbar. Dazu gehören zum einen:

Informationsobjekt	Informationsgehalt/Nutzen
`HttpSession`	Objekt- und Datenspeicher für die Lebensdauer einer Sitzung eines Anwenders
`ServletContext`	Die technische Bezeichnung einer im Webserver laufenden Webanwendung. Alle Servlet*instanzen*, die im selben Context laufen, sind einer einzigen Webapplikation zugeordnet und greifen auf dieselben geteilten Informationen zurück. Es ist auch möglich dasselbe Servlet in verschiedenen Webanwendungen zu nutzen. Dabei wird dann jeder Anwedung eine *Instanz* dieses Servlets zugeordnet. Des Weiteren stellt der `ServletContext` einen `Request Dispatcher` bereit, den u.a. auch Struts benutzt um einen Request zu bearbeiten und auf die nächste Seite zu verzweigen. Ebenfalls verfügbar sind Informationen zur Laufzeitumgebung, in der sich die Servlet*instanz* befindet.
`ServletConfig`	Metadatenspeicher für ein Servlet oder anderes ausgedrückt: Objektrepräsentation einer Servlet Konfiguration auf der Basis einer Konfigurationsdatei einer Webanwendung wie z.B. `web.xml`. Inhalte: • `<init>` Konfigurationsparameter und deren Werte • der Servlet Name • Zugriff auf den `ServletContext` Das `ServletConfig` Objekt wird einer Servletinstanz in der `init(ServletConfig)` Callback Methode bereitgestellt.

Tabelle 2: Die Bedeutung der Servlet API Objekte `HttpSession`, `ServletContext` und `ServletConfig`

Zum anderen ermöglicht es der `PageContext` dem Entwickler seine eigenen Informationen, bzw. Objekte dort zu hinterlegen. Z.B. bei der Implementierung eigener Tags. In Servlets oder Struts `Action` Klassen speichert er seine Attribute im bereits von der Servlet - Engine oder dem Webserver erzeugten und mitgelieferten `Request` Objekt, oder wahlweise in dessen zugeordneter `Session`. Im Rahmen von Struts wird dies weitesgehend durch die Modelklassen, den `ActionForm`s erledigt.

Innerhalb einer selbstimplementierten Tag Klasse steht dem Entwickler außer dem `Session` Objekt weder ein Request Objekt noch eine `ActionForm` zur Verfügung. Es fehlt also z.B. die Möglichkeit request bezogene Daten, die auch sinnvollerweise nur diese Lebensdauer haben sollen, zu speichern. Stattdessen das `HttpSession` Objekt zu benutzen, kann je nach Häufigkeit und Größe der Anwendung unerwünschte Folgen haben, wie z.B. einen Absturz der Anwendung oder des Servers. Wenn all diese Daten die Lebensdauer einer Session haben, kann im Falle eines Hackerangriffs, der Server mit Massenmüll an Daten in die Knie gezwungen werden.

Auf der anderen Seite ist es durch diese Praktik auch viel schwieriger, die Anwendung zu warten oder weiterzuentwickeln, da die gespeicherten Informationen unstrukturiert nur unter einem oftmals auch noch hart codierten Namen an einer Stelle gespeichert werden, ohne die

Möglichkeit, den Sinn und die eindeutige Zugehörigkeit der Daten zu einem Anwendungsbereich z.B. Benutzerverwaltung rekonstruieren zu können.

Um Daten mit jeder im jeweiligen Anwendungsfall sinnvollen Lebensdauer zu speichern, bietet der `PageContext` alle Möglichkeiten, d.h. dort hinterlegte Objekte können den `page`, `request`, `session` und `application` Gültigkeitsbereich zugewiesen bekommen.

Zurück zur Klasse `RequestUtils` und deren Methode `RequestUtils.message(PageContext, String bundle, String locale, String key, Object args[])`. Zuerst muss das richtige `MessageResources` Objekt unter Auswertung des `bundle` Parameters sofern gesetzt, gefunden werden. Hierfür wird die Methode `RequestUtils.retrieveMessageResources(PageContext, String bundle, boolean checkPage Scope)` aufgerufen. Diese Methode untersucht, ob im `PageContext` unter dem Schlüssel `bundle` ein `MessageResources` Objekt verfügbar ist. Gesucht wird in jedem Lebensdauerbereich, also von `PageContext.PAGE_SCOPE` bis `PageContext.APPLICATION_SCOPE`.

Sofern der Entwickler die Nameskonventionen eingehalten hat, d.h. entweder in der `web.xml` ein Element

```
<init-param>
      <param-name>application</param-name>
      <param-value>[z.B. ApplicationResources]</param-value>
<init-param>
```

spezifiziert oder in der `struts-config.xml` ein Element

```
<message-resources parameter=" com.test.one.ApplicationResources"/>
```

definiert hat, liefert die Methode das gewünschte `MessageResources` Objekt zurück. Nun muss noch die `Locale`, also die richtige Sprache des auf der JSP darzustellenden Labels ermittelt werden. Dazu wird die Methode `RequestUtils.retrieveUserLocale(PageContext, String locale)` aufgerufen. Über den `PageContext` wird über dessen `Session` und falls nicht vorhanden im `Request` nach einem `Locale` Objekt gesucht. Da im `org.apache.struts.taglib.bean.MessageTag` Objekt das locale String Attribut sofern nicht überschrieben mit dem default Wert der Konstanten `Globals.LOCALE_KEY` initialisiert ist und an `RequestUtils.message(PageContext, String bundle, String locale, String key, Object args[])` übergeben wird, kann immer ein `Locale` Objekt zurückgegeben werden.

Da nun alle Informationen vorliegen, also `MessageResources` und `Locale` kann schließlich die entscheidende Methode `MessageResources.getMessage(Locale l, String key, Object[] args)` aufgerufen werden. Wie schon erwähnt werden die Labels mit ihren Werten und allen Sprachvarianten in der Klasse `org.apache.struts.util.PropertyMessageResources` in einer `java.util.HashMap` gespeichert. Die `getMessage()` Methode greift auf diese `HashMap` zu und liefert den von Anfang an gesuchten Label, der auf die JSP kommt.

Schritt 2: `ActionServlet.initOther()`

Nach `ActionServlet.initInternal()` von `ActionServlet.init()` werden mit `ActionServlet.initOther()` die `web.xml` Konfigurations `<init>` Parameter eingelesen. Dies geschieht mittels des `ServletConfig` Objekts und dessen Methode `getInitParameter(String)`. In `ActionServlet.initOther()` sind dies die reservierten Parameter `config` und `debug`. In der `web.xml` stehen folgende Werte:

```
<init-param>
    <param-name>config</param-name>
    <param-value>/WEB-INF/struts-config.xml</param-value>
</init-param>
```

```
<init-param>
    <param-name>debug</param-name>
    <param-value>0</param-value>
</init-param>
```

Der config Parameter gibt an, dass im root Verzeichnis der Webapplikation unter WEB-INF die struts-config.xml liegt. Handelt es sich um eine Struts 1.1 Anwendung ist es die des Defaultmoduls. Der debug Parameter legt fest wie hoch der Debug Level ist. Ist er 0, werden alle Meldungen, auch die von Struts auf der Console ausgegeben. Die Werte für die Parameter config und debug werden als Attribute im ActionServlet gespeichert.

Schritt 3: ActionServlet.initServlet()

Im Mittelpunkt steht die Konfiguration eines org.apache.commons.digester.Digester für die web.xml. Zunächst werden die dtd Dateien für die web.xml und struts-config.xml registriert. D.h. die URLs der Dateien struts-config_1_0.dtd, struts-config_1_1.dtd, web-app_2_2.dtd, web-app_2_3.dtd werden dem Digester nacheinander übergeben. Mittels der Methode Digester.register(String publicid, String url) hinterlegt er die übergebenen Daten in einer HashMap als Quelle für dtd URLs, die auf dtd Dateien verweisen, die wiederum die Entitäten enthalten. Solche dtd Entitäten sehen beispielsweise aus der struts-config_1_1.dtd folgendermaßen aus:

```
<!ENTITY % BeanName "CDATA">
<!ENTITY % Boolean "(true|false|yes|no)">
<!ENTITY % ClassName "CDATA">
```

Diese werden in Ermangelung der Möglichkeit von Datentypdefinitionen in dtds eingesetzt, um Pseudodatentypen für Attribute von Elementen zu definieren. Z.B.

```
<!ELEMENT action (icon?, display-name?, description?, set-property*, exception*,
forward*)>
...
<!ATTLIST action         attribute       %BeanName;       #IMPLIED>
<!ATTLIST action         className       %ClassName;      #IMPLIED>
...
```

An dieser Stelle in der struts-config_1_1.dtd wird z.B. das <action> Element definiert, so wie es in einer struts-config.xml verwendet werden muss, d.h. welche Attribute gesetzt sein müssen und können.

Nach der Registrierung der dtds müssen die sogenannten processing rules für das Parsen der web.xml spezifiziert werden. Die ersten Informationen die aus diesem Deployment Descriptor geholt werden, betreffen das Controller Servlet der Webapplikation. In einer Struts Anwendung handelt es sich hier um das org.apache.struts.action.ActionServlet, oder einer davon abgeleiteten Klasse. Die Informationen, die diesbezüglich beschafft werden sind:

servlet-mapping	Der Name unter dem ein Servlet im Browser aufgerufen wird. Ist der hier eingetragene Name z.B. startServlet lautet die Webadresse auf localhost: http://localhost:8080/startServlet und um Browser erscheint die Seite, die aus dem HTML Quelltext des Servlets hervorgeht.
servlet-name	In einer Struts Anwendung ist der Name per Konvention action.
url-pattern	Ein Name für die Endung einer Webadresse. Z.B. http://localhost:8080/start.do In Strutsanwendungen ist die Extension per Konvention .do

Tabelle 3: Wichtige web.xml Attribute zur Konfiguration eines ActionServlets

Es wird also das `<servlet-mapping>` Element mit seinen Kindelementen ausgelesen. Damit der `Digester` diese Informationen auslesen kann, muss eine `org.apache.commons.digester.Rule` dafür erzeugt werden. Mittels der Methode `Digester.addCallMethod(String pattern, String methodName)` wird eine `org.apache.commons.digester.CallMethodRule` erzeugt. Die Methode `Digester.addCallMethod` wird für das Parsen für jedes xml Element zuerst aufgerufen. Der Parameter `pattern` repräsentiert das xml Element, das geparst werden soll. Vorrausgesetzt es gäbe ein xml Dokument mit folgendem Inhalt:

```
<a>
  <b>
    <c>xyz</<c>
  </b>
<a>
```

Um das richtige Element aufzurufen, wie z.B. das `<c>` Element, wird folgende Notation verwendet, die genauso auch für den `pattern` String Parameter von `Digester.addCallMethod()` verwendet wird: `a/b/c`

Es handelt sich um ein einfaches hierarchisches Muster, wobei jeder '/, eine Vertiefung der Hierarchieebene repräsentiert. In `Digester.addCallMethod(String pattern, String method Name)` wird ein `org.apache.commons.digester.CallMethodRule` Objekt erzeugt und dem `Digester` mittels seiner `addRule(String pattern, Rule rule)` Methode seinem Regelkatalog hinzugefügt. Diese Methode ordnet dem xml Element, durch den `pattern` Parameter repräsentiert, die richtige Regel zu, die benötigt wird, um die richtige Aktion ausführen zu können, wenn der Parseprozess beim anschließenden Parsen bei diesem Element angelangt ist. Der Regelkatalog ist ein Objekt der Klasse `org.apache.commons.digester.RulesBase`, das in einer `java.util.ArrayList` alle `org.apache.commons.digester.Rule` Objekte speichert.

Um später die zwei Kindelemente von `<servlet-mapping>` zu speichern, müssen ebenfalls zwei der dafür zuständigen Regel `org.apache.commons.digester.CallParamRule` dem `Digester` hinzugefügt werden. Dafür benutzt das `ActionServlet` die Methode `Digester.addCallParam( String pattern, int paramIndex)`. Diese erzeugt dann ein `CallParamRule` Objekt und fügt es ebenfalls mittels `Digester.addRule(String pattern, Rule rule)` dem Regelkatalog `RulesBase` unter dem erforderlichen xml Element hinzu.

Um die `web.xml` als Objektrepräsentation parsen zu können wird über den `ServletContext` mit seiner Methode `getResourceAsStream(„/WEB-INF/web.xml")` ein `java.io.InputStream` besorgt, das mittels `Digester.parse(InputStream)` geparst wird. Hierzu ruft das `ActionServlet` die Methode `Digester.parse(InputSource)` auf. Da es sich bei dem `Digester` um einen Ergeignis gesteuerten SAX Parser handelt, werden bei jedem Element auf das der Parseprozess stößt, u.a. folgende wesentliche Callbackmethoden aufgerufen.

- `startElement(String namespaceURI, String localName,String qName, Attributes list)`
- `endElement(String namespaceURI, String localName, String qName)`

Nachdem nun dem `Digester` alle notwendigen Regeln aus der `web.xml` für das `ActionServlet` gegeben wurden kann der Parseprozess gestartet werden.

Generell wird in diesen zwei Methoden die Applikationslogik implementiert, die das ausführen, was passieren soll, wenn der Parser am Beginn bzw. am Ende eines bestimmten Elements angelangt ist. Beim `Digester` geht es immer darum, für das jeweilige Element in seiner entsprechenden Hierarchiestufe die richtigen `Rules` unter Angabe des Elementpatterns, also s.o. z.B. `a/b/c`, zu holen und auszuführen. D.h. für eine gefundene Rolle wird dessen Callbackmethode `Rule.begin(org.xml.sax.AttributeList)` aufgerufen und somit die entsprechende Funktionalität, die hinter diesem Element steht, muss zur Anwendung gebracht werden. Das

entsprechend selbe passiert wenn die `endElement()` Methode des `Digesters` aufgerufen wird. Hier kommt die `Rule.end()` zur Ausführung.

Damit der `Digester` immer die richtige Hierarchieebene des Elements kennt, reicht natürlich nicht der Name des Elements allein. Aus diesem Grund wird immer der Pfad des übergeordneten Elements in einem globalen `String` Attribut `match` gesichert. So wird dieses Attribut bei jedem Kindelement in der `Digester.startElement()` um einen '/, und den Namen des Kindelements erweitert. Entsprechend wird das Attribut in der `Digester.endElement()` Methode reduziert.

Schritt 4: `ActionServlet.initModuleConfig(String prefix, String paths)`

Der vierte Schritt ähnelt in seiner Zielsetzung dem vorherigen. Ziel ist es auch hier eine xml Datei zu parsen und deren Inhalte applikationsweit zur Verfügung zu stellen. Dazu wird hier auch ein `Digester` konfiguriert und dieser parst anschließend das xml Javaobjekt. Diesmal sind es ein oder mehrere `struts-config.xml` deren Inhalte in einem `org.apache.struts.config.ModuleConfig` Objekt zu hinterlegen sind.

Genauso wie bei der Erzeugung eines `MessageResources` Objekts wird auch ein `ModuleConfig` Objekt mit einer `ModuleConfigFactory` hergestellt. Da auch diese Fabrik wie ihr Pendant `MessageResourcesFactory` abstrakt ist, liefert die Struts Distribution auch hier eine Standardimplementierungsfabrik `DefaultModuleConfigFactory`, die die Methode `ModuleConfig.createModuleConfig(String prefix)` aus ihrer Superklasse implementiert und ein inhaltsleeres `org.apache.struts.config.impl.ModuleConfigImpl` Objekt erzeugt und zurückliefert. Dieses enthält `java.util.HashMaps` und `java.util.ArrayLists` die später vom `Digester` mit den Daten aus einer `struts-config.xml` zu füllen sind.

Die Initialisierung des `Digester`'s für `ModuleConfig` Objekte übernimmt die Hilfsmethode `ActionServlet.initConfigDigester()`. Ein deutlicher Unterschied zur Initialisierung in `ActionServlet.initServlet()` ist, dass ein ganzes `RuleSet` in Form eines `org.apache.struts.config.ConfigRuleSet` Objekts mittels `Digester.addRuleSet(RuleSet)` als Konfigurationsgrundlage des `Digesters` gesetzt wird und nicht nur manuell Rollen zum parsen des `web.xml` Elements `web-app/servlet-mapping` spezifiziert werden.

Neben den vordefinierten Standardregeln, die die Struts 1.1 Distribution vom commons-digester Package her nutzt, hat der Entwickler die Möglichkeit eigene processing rules zu implementieren, sie in der `web.xml` zu deklarieren und sie in den Parseprozess des `Digester` zu integrieren. Die Deklaration schreibt vor, durch Komma getrennte vollklassifizierende Klassennamen zu notieren, deren Implementierungen alle von der Klasse `org.apache.commons.digester.RuleSet` erben müssen. Dieser String muss in der `web.xml` folgendermaßen eingetragen werden:

```
<init-param>
    <param-name>rulesetes</param-name>
    <param-value>com.test.RuleSetImpl1,com.test2.RuleSetImpl2</param-value>
<init-param>
```

So kann das `ActionServlet` über `ServletConfig.getInitParameter(„rulesets")` den Klassenstring abfragen, die einzelnen Klassennamen extrahieren und die Klassen laden. Das Laden geschieht wiederum durch `RequestUtils.applicationInstance(String clazz)` und das resultierende `RuleSet` wird mit `Digester.addRuleSet(RuleSet)` eingebunden und somit beim Parsen der `struts-config.xml` berücksichtigt. Man könnte mit anderen Worten sagen: der gesamte Digester – Parse – Mechanismus ist plug-in fähig, sehr flexibel, weil beliebig erweiterbar, natürlich über das Parsen von `web.xml`- und `struts-config.xml` Dateien hinaus.

Der `Digester` ist nun soweit konfiguriert, dass das Parsen aller `struts-config.xml` Dateien, die durch Komma getrennt, in dem `paths` Parameter aus `ActionServlet.initModuleConfig( String prefix, String paths)` enthalten sind, beginnen kann. Der erste Parameter `prefix` ist leer, da die Methode erstens zum ersten Mal für das Defaultmodul aufgerufen wurde und zweitens weil die Subapplikationen bei denen der `prefix` Parameter nicht leer sein darf, erst nach dem Initialisieren der Defaultmoduls an die Reihe kommen.

Das Parsen der `struts-config.xml` des Defaultmoduls wird mit der Methode `parseModule ConfigFile(String prefix, String paths, ModuleConfig, Digester, String path)` eingeleitet. Dazu wird wie beim Parsen der `web.xml` ein `InputSource` Objekt erzeugt, indem über den `path` Parameter, der sich aus einem (Null)Prefix und der einem `struts-config.xml` Dateinamen darangehängt zusammensetzt. Mittels `ServletContext().getResourceAsStream(String path)` wird zusätzlich ein `InputStream` Objekt geholt und die `InputSource` mit diesem ByteStream geladen. Der letztendliche Startschuss für das Parsen ist das Statememt `Digester.parse(Input Stream)`.

Schritt 5: Der Parseprozess der struts-config.xml

Der Parseprozess wird anhand des Erzeugens des Javarepräsentationsobjekten des klassischen `struts-config.xml` Elements `<form-bean>` betrachtet. Um dies zu ermöglichen, wird in der Methode `Digester.addRuleSet(RuleSet)` die Defaultmethode `addRuleInstances(Digester)` des Interfaces `RuleSet` aufgerufen. Das für `struts-config.xml` Elemente spezialisierte `RuleSet` `org.apache.struts.config.ConfigRuleSet` regelt die Erzeugung dieser Elemete in drei Schritten. Zunächst die Erzeugung einer Regel für ein `<form-bean>` Element:

1. <u>Erzeugung einer `org.apache.commons.digester.ObjectCreateRule`</u>

   Die Methode `Digester.addObjectCreate(String pattern, String className, String attributeName)` übergibt dem `ObjectCreateRule` Konstruktor den vom `ConfigRule Set` spezifizierten Wert `org.apache.struts.action.ActionFormBean` für den Parameter `className` (`ObjectCreateRule(className, attributeName)`). Mittels dieses Parameters weiss die `ObjectCreateRule`, dass sie diese von `org.apache.struts.config.FormBean Config` abgeleitete Klasse erzeugen soll. Der zweite mit „`className`" initialisierte Parameter `attributeName` ist nicht von entscheidender Bedeutung. Er ist optional und definiert einen Klassennamen für eine abgeleitete Klasse, und wird nur dann benutzt, wenn die zu erzeugende Klasse dieses Attribut besitzt. Der `pattern` Parameter der `Digester.addObject Create()` Methode bestimmt das Element der `struts-config.xml` an der diese Regel vom `Digester` angewendet werden muss. Bei einem `<form-bean>` Element ist dies bekanntermaßen: `struts-config/form-beans/form-bean`

   Wann immer also der Parseprozess des `Digester`s auf dieses Element stößt wird die Callbackmethode `Digester.startElement(String namespaceURI, String localName, String qName, Attributes list)` aufgerufen. Der `Digester` holt sich anhand von `namespaceURI`, hier `form-bean` und seinem globalen `match` Attribut, das ja die gegenwärtige Position innerhalb des xml Baumes widerspiegelt, aus seinem Regelkatalog `org.apache. commons.digester.RulesBase` die für das `form-bean` Element korrespondierenden Regeln und führt diese aus. D.h. er ruft deren Callbackmethoden `Rule.begin()` auf. Am Ende dieser Methode wird das erzeugte `ActionFormBean` Objekt auf dem Element Stack des `Digester` mittels `Digester.push(Object)` gelegt. In der Callbackmethode `Digester.endElement()` wird `Rule.end()` aufgerufen.

   Auf diese Weise werden die für die Erzeugung eines `org.apache.struts.action. ActionFormBean` Objekts, welches ein `form-bean` Element repräsentiert, erforderlichen `Rules`

ausgeführt und in ein `ModuleConfig` Objekt gespeichert, um zur Laufzeit ein `ActionForm` Objekt zu erzeugen, das der Entwickler in der `struts-config.xml` spezifiziert hat unter den Attributen eines `<form-bean>` Elements angegeben hat.

2. Erzeugung einer `org.apache.commons.digester.SetPropertiesRule`

   Diese Regel ist für das Setzen der Attribute des in 1. erzeugten Elementrepräsentationsobjekts zuständig. Die notwendigen Daten werden von der Callbackmethode `Digester.startElement()` in einem `org.xml.sax.Attributes` Objekt geliefert. Analog zu 1. steht in `ConfigRuleSet.addRuleInstances(Digester)` als nächste Anweisung die Erzeugung der `SetPropertiesRule` mit dem Wert `struts-config/form-beans/form-bean`, der wieder im `pattern` Parameter von `Digester.addSetProperties(String pattern)` übergeben wird. Das zugehörige Element `ActionFormBean` wird über den Element Stack des Digesters mittels `Digester.peek()` geholt und es bedarf deshalb keiner Logik für eine Objektsuche. Die Attribute und Werte aus dem `org.xml.sax.Attributes` Objekt werden in einer `HashMap` gespeichert und mit dem `org.apache.struts.action.ActionFormBean` Objekt aus 1. an die Utilityklasse `BeanUtils` und dessen Hilfsmethode `BeanUtils.populate(Object, Map)` weitergegeben. Diese umfangreiche und komplexe Methode sucht für jedes Attribut – Wert – Paar eine entsprechende `set` Methode der übergegeben Beanklasse aus dem Parameter `Object`.

   Ergebnis: das Elementrepräsentationsobjekt `org.apache.struts.action.ActionFormBean` des Elements `<form-bean>` enthält nun all dessen Attribute aus der `struts-config.xml`. Jetzt muss es noch in das `ModuleConfig` Objekt integriert werden.

3. Erzeugung einer `org.apache.commons.digester.SetNextRule`

   Die Integration eines jeden Elementrepräsentationsobjekts wird von dieser `org.apache.commons.digester.Rule` durchgeführt. Wie in den beiden vorangegangenen `Rule`s ist auch hier der `pattern` Paramater wieder `struts-config/form-beans/form-bean`. Der zweite Parameter gibt die Methode des Zielobjektes an, mit der das Elementrepräsentationsobjekt im Zielobjekt gespeichert werden soll. Der Parameter `paramType` mit dem Wert `org.apache.struts.config.FormBeanConfig` (Basisklasse von `ActionFormBean`) gibt an, als welcher Typ das Elementrepräsentationsobjekt im `ModuleConfig` Objekt gespeichert werden soll.

   Das Zielobjekt ist das `ModuleConfig` Objekt und dessen aufzurufende Methode `ModuleConfig.addFormBeanConfig(org.apache.struts.config.FormBeanConfig)`. In `SetNextRule.end()` (die `begin()` Methode ist leer) werden zuerst zwei Objekte vom Stack des `Digester` geholt: `ModuleConfig` und das `ActionFormBean`. Anschließend werden diese beiden Objekte mit den Übergabeparametern `methodName` und `paramType` an `MethodUtils.invokeMethod(Object, String methodName, Object[] args, Class[] parameterTypes)` weitergegeben. Diese Methode wiederum benutzt eine Hilfsmethode `MethodUtils.getMatchingAccessibleMethod(Class, StringmethodName, Class[] parameterTypes)`, um die Methode `ModuleConfig.addFormBeanConfig(FormBean Config)` als `java.lang.reflect.Method` Objekt zu holen. Als letzter Schritt wird dieses `Method` Objekt mit `Method.invoke(Object caller, Object[] args)` ausgeführt. `Object caller` ist `ModuleConfig`, `args` ein einelementiger Array mit dem `org.apache.struts.action.ActionFormBean` zu einem gecasteten `org.apache.struts.config.FormBeanConfig` Objekt.

   Auf diesem Wege ist ein `<form-bean>` Element zu einem `FormBeanConfig` Objekt im Module Config Objekt gespeichert worden und steht somit applikationsweit zur Verfügung.

Dieser drei schrittige Mechanismus wiederholt sich nicht nur für jedes andere `<form-bean>` Element, sondern auch für andere `struts-config.xml` Elemente wie `<action>`, `<message-resources>` und `<forward>` nach dem selben Prinzip.

Schritt 6: `ActionServlet.initModuleMessageResources(ModuleConfig)`

Da nun der gesamte Inhalt der `struts-config.xml` des Defaultmoduls in das `ModuleConfig` Objekt überführt wurde, sind die `MessageResourcesConfig` Objekte mittels `ModuleConfig.find MessageResourcesConfigs()` verfügbar. Mit deren Attribut `parameter`, das den vollklassifizieenden Pfadnamen beinhaltet, lässt sich genauso wie in Schritt 1 über die bekannten Fabrikklassen die deklarierten `MessageResources` Objekte erzeugen. Diese werden zur applikationsweiten Verfügarkeit mit `ServletContext.setAttribute(MessageResourcesConfig.getKey(), Message esources)` global gespeichert. Jetzt kann auf den JSP Seiten auf deren Labels per Schlüssel abgerufen werden.

Schritt 7: `ActionServlet.initModuleDataSources(ModuleConfig)`

Sofern in der `struts-config.xml` `<data-source>` Elemente definiert wurden, liefert `ModuleConfig.findDataSourceConfigs()` `DataSourceConfig` Objekte zurück. Mittels `Request Utils.applicationInstance(DataSourceConfig.getType())` werden `DataSource` Objekte erzeugt, die den Zugang zu Datenbanksystemen vereinfachen. Der `type` Parameter von `Data ourceonfig` ist sofern nicht angegeben, auf `org.apache.struts.util.GenericDataSource` gesetzt, einer Defaultimplementierung des Interfaces `javax.sql.DataSource`. Die Attribute von `GenericDataSource` werden aus den Werten des `<data-source>` Elements durch die Hilfsmethode `BeanUtils.populate(Object, Map)` initialisiert.

Diese Methode, bzw. die Klasse `org.apache.commons.beanutils.BeanUtils` dient zur Initialisierung von Attributen beliebiger Objekte. Die Reflection Mechanismen sind wiederum in der Utilityklasse `PropertyUtils` untergebracht, die Datentypkonvertierungsdienste in `Convert Utils`.

Nach der Initialisierung der Attribute der `GenericDataSource` Objekte werden sie wie in Schritt 6 mit `ServletContext.setAttribute(DataSourceConfig.getKey(), DataSource)` zur globalen Verfügbarkeit gesichert.

Schritt 8: `ActionServlet.initModulePlugIns(ModuleConfig)`

Sofern in der `struts-config.xml` `<plug-in>` Elemente definiert wurden, liefert `Module Config.findPlugInConfigs()` `PlugInConfig` Objekte zurück. Mittels `RequestUtils. applicationInstance(PlugInConfig.getType())` werden `PlugIn` Objekte erzeugt. Für jedes `PlugIn` werden auf die bekannte Art und Weise dessen Attribute gesetzt. Mit der Initialisierung `PlugIn.init(ActionServlet servlet, ModuleConfig config)` ist dieser Arbeitsschritt beendet.

Schritt 9: `Initialisierung der Subapplikationsmodule`

Dieser Vorgang umfasst die Initialisierung aller Subapplikationsmodule der Webanwendung. D.h. für jede Subapplikation muss aus deren eigenen `struts-config.xml` ein `ModuleConfig` Objekt erzeugt werden. Die erforderlichen Konfigurationsdaten werden mittels `Servlet Config().getInitParameterNames()` beschafft. Der Ablauf der Initialisierung ist genau derselbe wie für das Defaultmodul, Schritt 4 bis 8 wiederholen sich also.

Der einzige kleine Unterschied ist das sogenannte Prefix. Beim Defaultmodul ist dies ein Leerstring, bei einem Submodul muss dieser je Modul in der `web.xml` samt der des Namens der `struts-config.xml` Datei, die nicht notwendigerweise so heißen muss, angegeben werden:

```
<init-param>
  <param-name>config/examples</param-name>
  <param-value>/WEB-INF/struts-examples-config.xml</param-value>
</init-param>
<init-param>
  <param-name>config/test</param-name>
  <param-value>/WEB-INF/struts-tests-config.xml</param-value>
</init-param>
```

Es gibt in dieser Struts Applikation zwei Subapplikationen. Das Framework initialisiert jede unter Verwendung der Prefixes `/examples` und `/test` mit ihrer zugehörigen `struts-config.xml`. Der Wert `config` vor dem `/` ist genauso wie beim Defaultmodul Vorraussetzung beim parsen der `web.xml` durch das `ActionServlet`.

1.4 Das Request Processing

Dieses Kapitel befasst sich nachdem die Initialisierung abgeschlossen, d.h. das Struts Framework und die Webanwendung gestartet wurden, mit dem Ablauf der Requestbearbeitung, wenn der Benutzer durch Klick auf einen Link oder eine Submit Schaltfläche eine Aktion ausgelöst hat. Es wird untersucht, wie das Framework die richtige Aktion ermittelt, die notwendigen Ressourcen aquiriert und die Aktionsbearbeitung dem richtigen Submodul oder Defaultmodul zuweist.

Ein Blick auf den Stack Trace nach dem Ausführen einer Benutzeraktion mit anschließender absichtlich erzeugten Exception verrät schon einiges:

```
java.lang.NullPointerException

at com.tsystems.struts.util.db.WebTechConnection.isClosed(WebTechConnection.java:30)
at com.tsystems.struts.util.db.WebTechConnection.prepareStatement(WebTechConnection.java:59)
at com.tsystems.struts.util.db.StatementFacade.getStatement(StatementFacade.java:24)
at com.tsystems.struts.user.db.UserDBAdapter.findByUserNameAndPassword(UserDBAdapter.java:26)
at com.tsystems.struts.user.logic.UserHome.findByUserNameAndPassword(UserHome.java:16)
at com.tsystems.struts.user.struts.UserLoginAction.execute(UserLoginAction.java:32)
at org.apache.struts.action.RequestProcessor.processActionPerform(RequestProcessor.java:484)
at org.apache.struts.action.RequestProcessor.process(RequestProcessor.java:274)
at org.apache.struts.action.ActionServlet.process(ActionServlet.java:1482)
at org.apache.struts.action.ActionServlet.doPost(ActionServlet.java:525)
at javax.servlet.http.HttpServlet.service(HttpServlet.java:760)
at javax.servlet.http.HttpServlet.service(HttpServlet.java:853)
at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:247)
at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:193)
at org.apache.catalina.core.StandardWrapperValve.invoke(StandardWrapperValve.java:243)
at org.apache.catalina.core.StandardPipeline.invokeNext(StandardPipeline.java:566)
at org.apache.catalina.core.StandardPipeline.invoke(StandardPipeline.java:472)
at org.apache.catalina.core.ContainerBase.invoke(ContainerBase.java:943)
at org.apache.catalina.core.StandardContextValve.invoke(StandardContextValve.java:190)
at org.apache.catalina.core.StandardPipeline.invokeNext(StandardPipeline.java:566)
at org.apache.catalina.valves.CertificatesValve.invoke(CertificatesValve.java:246)
at org.apache.catalina.core.StandardPipeline.invokeNext(StandardPipeline.java:564)
at org.apache.catalina.core.StandardPipeline.invoke(StandardPipeline.java:472)
at org.apache.catalina.core.ContainerBase.invoke(ContainerBase.java:943)
at org.apache.catalina.core.StandardContext.invoke(StandardContext.java:2347)
at org.apache.catalina.core.StandardHostValve.invoke(StandardHostValve.java:180)
at org.apache.catalina.core.StandardPipeline.invokeNext(StandardPipeline.java:566)
at org.apache.catalina.valves.ErrorDispatcherValve.invoke(ErrorDispatcherValve.java:170)
at org.apache.catalina.core.StandardPipeline.invokeNext(StandardPipeline.java:564)
at org.apache.catalina.valves.ErrorReportValve.invoke(ErrorReportValve.java:170)
at org.apache.catalina.core.StandardPipeline.invokeNext(StandardPipeline.java:564)
at org.apache.catalina.valves.AccessLogValve.invoke(AccessLogValve.java:468)
at org.apache.catalina.core.StandardPipeline.invokeNext(StandardPipeline.java:564)
at org.apache.catalina.core.StandardPipeline.invoke(StandardPipeline.java:472)
at org.apache.catalina.core.ContainerBase.invoke(ContainerBase.java:943)
at org.apache.catalina.core.StandardEngineValve.invoke(StandardEngineValve.java:174)
at org.apache.catalina.core.StandardPipeline.invokeNext(StandardPipeline.java:566)
at org.apache.catalina.core.StandardPipeline.invoke(StandardPipeline.java:472)
at org.apache.catalina.core.ContainerBase.invoke(ContainerBase.java:943)
at org.apache.catalina.connector.http.HttpProcessor.process(HttpProcessor.java:1027)
at org.apache.catalina.connector.http.HttpProcessor.run(HttpProcessor.java:1125)
at java.lang.Thread.run(Thread.java:534)
```

Abbildung 3: Exception nach einer Benutzerinteraktion

Nachdem der Request alle Tomcat Webserver `Valve`s und `Pipeline`s durchlaufen hat wird er zum Schluss an das `ActionServlet.service()` aus der Basisklasse `HttpServlet` übergeben. Dies

bildet den Ausgangspunkt der Untersuchungen. Die vom Tomcat erzeugten und bearbeiteten Request und Response Objekte werden der Methode `ActionServlet.process(HttpServlet Request, HttpServletResponse)` übergeben.

Die folgenden Aufgaben zur Requestbearbeitung lassen sich in zwei große Blöcke einteilen. Schritt eins beschreibt die Vorbereitungsarbeit des `ActionServlet`. Schritt zwei analysiert die Umsetzung durch den `RequestProcessor`.

Schritt 1: Initialisierung des RequestProcessor

Anhand des übergebenen `Request` muss zuerst ermittelt werden zu welcher Subapplikation, dieser gehört. Ziel ist es, das richtige `ModuleConfig` Objekt zu finden und dem Request mittels `HttpServletRequest.setAttribute(String, Object)` zu übergeben. Dazu wird aus der bekannten Utilityklasse `RequestUtils` die Methode `selectModule(HttpServletRequest, Servlet Context)` aufgerufen. Entscheidend für die Identifikation des richtigen Moduls ist das `prefix` (siehe Schritt 4 aus Szenario 1 in diesem Kapitel). Das `prefix` legt der Applikationsentwickler in der `web.xml` in einem `<init>` Element fest. Das `prefix` des Defaultmoduls ist ein Leerstring, vor dem der für prefixes festgelegte Standardwert `config` steht. Bei einem Submodul namens `"submodule"` sieht dies im Vergleich zum Defaultmodul in der `web.xml` so aus:

```
<web-app>
  ...
  <servlet>
  ...
    <!-- Defaultmodul -->
    <init-param>
      <param-name>config</param-name>
      <param-value>/WEB-INF/struts-config.xml</param-value>
    </init-param>

    <!-- Submodul submodule -->
    <init-param>
      <param-name>config/submodule</param-name>
      <param-value>/WEB-INF/struts-submodule-config.xml</param-value>
    </init-param>

    <!-- Submodul test -->
    <init-param>
      <param-name>config/test</param-name>
      <param-value>/WEB-INF/struts-test-config.xml</param-value>
    </init-param>
    ...
  </servlet>
  ...
</web-app>
```

Bei einer Subapplikation folgt nach dem festgelegten Standardwert `config` ein `/` gefolgt von einem beliebigen Namen, hier `"submodule"`, dem sogenannten `prefix`. Dieses ermittelt `RequestUtils.selectModule()` zuerst durch den Aufruf von `RequestUtils.getModuleName( HttpServletRequest, ServletContext)`, die das `prefix` zurückliefert. Um das `prefix` zu ermitteln, geht die Methode folgendermaßen vor:

Zuerst werden alle in der `web.xml` verfügbaren Prefixes mittles `RequestUtils.getModule Prefixes(ServletContext)` besorgt. Diese Hilfsmethode ermittelt über `ServletContext. getAttributeNames()` generell alle gespeicherten Attribute. Bei der Initialisierung der übergebenen `ModuleConfig` Objekte in der Methode `ActionServlet.parseModuleConfigFile(String prefix, String paths, ModuleConfig, Digester, String path)` wurde während der Initialisierungsphase des Struts Framework jedes `ModuleConfig` Objekt mit **`ServletContext`**`.set`

Attribute(**Globals.MODULE_KEY** + **prefix**, ModuleConfig) gesichert. Globals.MODULE_KEY ist ein globales String Attribut mit dem Wert org.apache.struts.action.MODULE aus dem Interface org.apache.struts.Globals.

Der prefix Parameter stammt von der Methode ActionServlet.init() und wurde aus dem Enumeration Objekt von ServletConfig.getInitParameterNames() extrahiert, also direkt aus den <init-param> Elementen der web.xml. In ActionServlet.getModulePrefixes(ServletContext) werden aus dem ServletContext alle Attribute ausgelesen, deren Werte mit Globals.MODULE_KEY beginnen. Z.B. würden aus dem Ausschnitt der obigen web.xml die Werte config/submodule und config/test herauskommen. Die Werte nach config/ werden ausgeschnitten und über dem String[] Rückgabewert der aufrufenden Methode RequestUtils.getModuleName(String matchPath, ServletContext context) zur Verfügung gestellt.

Im nächsten Schritt kommt der matchPath Parameter ins Spiel. Bevor seine Bedeutung erklärt wird, zuerst sein Zustandekommen: die Methode RequestUtils.getModuleName(HttpServletRequest, ServletContext) ermittelt diesen Wert über HttpServletRequest.getServletPath(). Die Stringrückgabewerte dieser Methode setzen sich in einer Struts 1.1 Webapplikation immer so zusammen: /[module_name]/[action_name].do. module_name ist der Name der Subapplikation, also das jeweilige Modul. action_name repräsentiert aus einem bestimmten <action> Element aus der einer struts-config.xml den Wert des path Attributs. Z.B. <action path=**"/login"** ...>. Die Extension .do stammt aus servlet-mapping/url-pattern Element der web.xml. Beispielhaft kann ein servletPath, also in der Methode RequestUtils.getModuleName(...) die matchPath Variable folgenden Wert haben: /submodule/login.do.

Die Methode RequestUtils.getModuleName(String matchPath, ServletContext) untersucht nun anhand des matchPath Parameters und den Prefixes aus der String Array prefixes das richtige Modul für den eingegeangenen Request. Zuerst muss aus dem matchPath Parameter der Modulname extrahiert werden. Bei dem letzten Beispiel muss also aus /submodule/login.do der Wert submodule ausgeschnitten werden. Darauffolgend wird dieser Wert mit allen Prefixwerten aus dem String Array von der Methode RequestUtils.getModulePrefixes(ServletContext) verglichen. Ist eine Übereinstimmung gefunden, ist das richtige Modul zu dem Request gehört gefunden.

Dieses gefunde Modulprefix wird der Methode RequestUtils.selectModule(String prefix, HttpServletRequest, ServletContext) übergeben. So wie das ModuleConfig Objekt in der Methode ActionServlet.parseModuleConfigFile(String prefix, String paths, ModuleConfig, Digester, String path) während der Initialisierungsphase mit **ServletContext**.setAttribute(Globals.MODULE_KEY + prefix, ModuleConfig) gespeichert wurde, wird es jetzt, da der prefix Parameter bekannt ist, mit ServletContext.getAttribute(Globals.MODULE_KEY + prefix) ausgelesen. Das für diesen Request und dieses Modul richtige ModuleConfig Objekt wird im finalen Schritt im request scope mittels **HttpServletRequest**.setAttribute(Globals.MODULE_KEY + prefix, ModuleConfig) zur weiteren Bearbeitung eingetragen. Analog wird zum ModuleConfig Objekt mit den MessageResources des jeweiligen Moduls verfahren.

Auch dieses wurde in der Methode ActionServlet.initModuleMessageResources() in den Servlet**Context** mittels ServletContext.setAttribute(MessageResourcesConfig.getKey() + ModuleConfig.getPrefix(), MessageResources) nach der Initialisierung global, unter dem Modulprefix, gespeichert. Aus dem ServletContext wird es nun wieder mit ServletContext.getAttribute(Globals.MESSAGES_KEY + prefix) beschafft, und wie das ModuleConfig Objekt in den Userrequest verlagert: HttpServletRequest.setAttribute(Globals.MESSAGES_KEY, MessageResources). Die Stringkonstante MESSAGES_KEY aus dem Interface org.apache.struts.Globals mit dem Wert org.apache.struts.action.MESSAGE wird für alle MessageResources

Objekte als Schlüssel verwendet wenn es in einem `ServletContext` oder einem `HttpServletRequest` als Attribut gespeichert wird.

Somit ist die Methode `RequestUtils.selectModule(HttpServletRequest request, ServletContext context)` innerhalb von `ActionServlet.process(HttpServletRequest request, HttpServletResponse response)` abgearbeitet, d.h. der Request kann dem für das Modul zuständige `RequestProcessor` weitergegeben werden. Der richtige `RequestProcessor` wird sinngemäß anhand des `ModuleConfig` Objekts des `HttpServletRequest` ermittelt. Dazu erzugt die threadsafe Methode `ActionServlet.getRequestProcessor(ModuleConfig)` ein `RequestProcessor` Objekt, falls es noch nicht schon in den `ServletContext` unter dem Schlüssel `Globals.REQUEST_PROCESSOR_KEY + ModuleConfig.getPrefix()` als Attribut hinzugefügt wurde.

Falls nicht wird standardmäßig über `RequestUtils.applicationInstance(String clazz)` eine `RequestProcessor` Instanz erzeugt. Den Wert des `clazz` Parameters liefert `ModuleConfig.getControllerConfig().getProcessorClass()`. Dieser ist per Default `org.apache.struts.action.RequestProcessor`. Nach der Instanziierung des `RequestProcessor`s wird er mit seiner `init(ActionServlet, ModuleConfig)` Methode initialisiert. In dieser Methode befindet sich keine nenneswerte Implementierung. Nach diesem Schritt wird er wie das `ModuleConfig` Objekt im `ServletContext` hinterlegt: `ServletContext.setAttribute(Globals.REQUEST_PROCESSOR_KEY + ModuleConfig.getPrefix(), RequestProcessor)`.

Mit dem Aufruf von `RequestProcessor.process(HttpServletRequest, HttpServletResponse)` endet die Zuständigkeit des `ActionServlet` seit Struts 1.1. Alles weitere bis zum Verzweigen auf die nächste JSP oder `org.apache.struts.action.Action` wird jetzt vom `RequestProcessor` durchgeführt.

Schritt 2: `RequestProcessor.process()`

Die zentrale Methode `RequestProcessor.process(HttpServletRequest, HttpServletResponse)` implementiert und kapselt mit ihren internen Hilfsmethoden die gesamte Logik der Requestbearbeitung bis zum Aufruf von `javax.servlet.RequestDispatcher.forward( HttpServletRequest, HttpServletResponse)`, der das Requestnavigationsziel, eine JSP oder eine andere `org.apache.struts.action.Action` bis hin zu deren JSP, zur Anzeige auf den Browser bringt.

Schritt 2.1: `RequestProcessor.processMultipart()`

Der erste Methodenaufruf. Es wird untersucht, ob der Content-Type des eingegangenen Request einfaches HTML ist, oder ob es sich um einen multipart Content-Type handelt. Ob also beispielesweise ein Word oder pdf Dokument über den Request versendet wurde. In diesem Zusammenhang wird auch die Requestmethode auf POST oder GET untersucht. Ist sie nicht POST wird der Request sofort unverändert zurückgegeben. Andernfalls wird der Request in ein `org.apache.struts.upload.MultipartRequestWrapper` Objekt verpackt.

Schritt 2.2: `RequestProcessor.processPath()`

Ziel dieser Methode ist es, vom Request URI den für die Initialisierung des ActionMapping Objekts richtigen Teil zu extrahieren. Dieser hat folgendes Format: /[webapp_main_dir]/[module_name]/[action_name].do. die Methode versucht den Request URI zuerst mittels HttpServletRequest.getAttribute(INCLUDE_PATH_INFO) und falls ohne Ergebnis über HttpServletRequest.getPathInfo() und HttpServletRequest.getAttribute(INCLUDE_SERVLET_PATH) auszulesen. Die „Source of last resort" ist HttpServletRequest.getServletPath() und das Ergebnis könnte z.B. sein: /submodule/**login**.do. Über Stringoperationen wird in diesem Fall der String /login augeschnitten und als Ergebnis zurückgegeben. Allgemein immer der Teilstring vom inklusive letzten / bis exklusive des ..

Schritt 2.3: RequestProcessor.processLocale()

Diese Methode hat die Aufgabe, ein Locale Objekt als Attribut in die Session des Requests zu speichern. Zuerst wird jedoch überprüft, ob dies auch im ModuleConfig Objekt so konfiguriert wurde. Dies wird mit ModuleConfig.getControllerConfig().getLocale() gecheckt, die true zurückgibt, wenn in der struts-config.xml das <controller **locale="true"** ...> Element konfiguriert wurde. Hat dies der Entwickler so spezifiziert, wird das HttpSession Objekt des HttpServletRequest besorgt. Befindet sich noch kein Locale Objekt in der HttpSession, d.h. wenn HttpSession.getAttribute(Globals.LOCALE_KEY) null ergibt, wird die Locale des HttpServletRequest verwendet, die immer durch den Servlet Contrainer bereitgestellt wird, und als Attribut in dessen HttpSession unter dem Defaultattribut Globals.LOCALE_KEY hinterlegt: HttpSession.setAttribute(Globals.LOCALE_KEY, Locale).

Schritt 2.4: RequestProcessor.processContent()

Eine sehr kleine Hilfsmethode, die in den Contenttype ModuleConfig.getControllerConfig().getContentType() in die HttpServlet**Response** über deren Methode setContentType(String) setzt.

Schritt 2.5: RequestProcessor.processMapping()

Anhand des von RequestProcessor.processPath() extrahierten path Parameters wird mit ModuleConfig.findActionConfig(path) das diesem Request zugehörige ActionMapping Objekt besorgt. Wie bei der einführenden Beschreibung der ActionMapping Klasse erwähnt, enthält diese das in der struts-config.xml unter einem <action> Element anzugebende path und type Attribut. Unter dem Wert von path wurde dieses ActionMapping Objekt in der ModuleConfig in der entsprechenden HashMap gespeichert und kann nun bequem ausgelesen und als Attribut im eingegangenen Request mittels HttpServletRequest.setAttribute(Globals.MAPPING_KEY, ActionMapping) eingetragen werden.

Es kann jedoch sein, dass im ModuleConfig Objekt unter dem path Schlüssel kein ActionMapping vorliegt. In diesem Fall gibt es evtl. noch einen Ausweg. Einem action Element der struts-config.xml ist gestattet das unknown Attribut auf true zu setzen. Nach diesem action Element sucht RequestProcessor.processMapping(), indem alle ActionConfig Objekte, durch ModuleConfig.findActionConfigs() zurückgeliefert, auf das Vorhandensein dieses einmaligen Attributs überprüft werden. Entspricht ein action Element dieser Spezifikation wird das repräsentierende ActionConfig zurückgeliefert, ansonsten null.

Schritt 2.6: RequestProcessor.processRoles()

Seit Struts 1.1 gibt es die Möglichkeit das Ausführen von `org.apache.struts.action.Action` Klassen durch Rollen zu schützen. Es ist ein Securitymechanismus, der über die `ActionMapping` Klasse genutzt werden kann.

Bei einem `action` Element lässt sich ein Attribut `roles` setzen. Es handelt sich dabei gemäß der dtd `struts-config_1_1.dtd` um einen Komma separierten String, der aus einfachen Namen besteht, von denen mindestens einer in `HttpServletRequest.isUserInRole(String)` true zurückliefern muss. D.h. dass der Benutzer berechtigt, ist diese Aktion auszuführen. Um dies zu prüfen holt diese Methode alle Rollennamen als String Array aus dem mitgelieferten `ActionMapping` Objekt mittels `ActionMapping.getRoleNames()`. Anschließend wird einfach geprüft, ob eines der String Array Elemente über `HttpServletRequest.isUserInRole(String)` vorhanden ist. Deklariert werden alle Zugriffbegrenzungen im Deployment Deskriptor `web.xml`. unter dem Element `<security-constraint>`.

Schritt 2.7: `RequestProcessor. processActionForm()`

Ziel der Methode: Die Instanziierung eines `org.apache.struts.action.ActionForm` Objektes, das anhand des eingegenagenen Request und des zuvor instanziierten `ActionMapping` und dessen Attribut `name`, dem Verweis auf den vollklassifizierenden Klassennamen unter dem `<form-bean>` Element ermittelt wird. Anschließend muss wie üblich noch je nach `Action Mapping.getScope()` entweder im `HttpServletRequest` oder dessen `HttpSession` als Attribut gespeichert werden: `setAttribute(ActionMapping.getAttribute(), ActionForm)`.

Die Instanziierung des `ActionForm` Objekts läuft wie die Erzeugung aller Objekte über die `RequestUtils` Hilfsklasse und deren Methode `RequestUtils.createActionForm(HttpServlet Request, ActionMapping, ModuleConfig, ActionServlet)`.

Schritt 2.8: `RequestProcessor.processPopulate()`

Nach dem `org.apache.struts.action.ActionForm` Objekt erzeugt wurde, müssen noch dessen Attribute mit den entsprechenden `set` Methoden initialisiert werden. Dies geschieht wie gewohnt mit der Variante `RequestUtils.populate(ActionForm, ActionMapping.getPrefix(), ActionMapping.getSuffix(), HttpServletRequest)` der `RequestUtils.populate()` Methoden. Diese Methode greift wiederum auf Services der Klasse `BeanUtils` zu, die die eigentliche Initialiserung via Reflection vornimmt.

Schritt 2.9: `RequestProcessor.processActionCreate()`

Wie das `ActionForm` Objekt muss auch die `Action` anhand `ActionMapping.getType()` erzeugt werden. Mit dem erhaltenen voll klassifizierenden Klassennamen der `Action` Klasse wird zunächst in der `Action HashMap` nach der richtigen Instanz gesucht. Falls nicht vorhanden wird sie mit `RequestUtils.applicationInstance(String)` erzeugt, in der `HashMap` gespeichert und zurückgegeben.

Schritt 2.10: `RequestProcessor.processActionPerform()`

Diese Methode dient lediglich zum Aufruf der `Action.execute()` Methode, die der Entwickler selbst implementiert hat. Am Ende der `Action.execute()` Methode schreibt der Entwickler `ActionMapping.findForward(...)`. Damit liefert `execute()` ein `ActionForward` Objekt zurück. Das richtige wird anhand des Wertes des `<forward name="...">` Attributs unter

dem jeweiligen `<action>` Element der `struts-config.xml` ermittelt. Entweder direkt von `ActionMapping.findForward(String)` oder über `ModuleConfig.findForwardConfig(String)`.

Schritt 2.11: **`RequestProcessor.processForwardConfig()`**

Im vorletzten Schritt muss zu Beginn der Request URI ermittelt werden. Dieser hat das bekannte Format: `/[module_name]/[action_name].do`. Vor dieser Ermittlung ist noch nicht bekannt, zu welchem Modul der Pfad von `ForwardConfig.getPath()` gehört. Er enthält entweder nur einen Pfad zu einer JSP oder zur einer `Action` jedoch ohne Moduleangabe. Den Pfad mit Modulprefix liefert `RequestUtils.forwardURL(HttpServletRequest, ForwardConfig)`. Dieser **Request URI** jetzt mit Modulprefix kann der Methode `doForward(String, HttpServletRequest, HttpServletResponse)` und damit dem letzten Arbeitsschritt übergeben werden.

Schritt 2.12: **`RequestProcessor.doForward()`**

In dieser Methode verwendet der `RequestProcessor` den `javax.servlet.RequestDispatcher`, den er über den `ServletContext.getRequestDispatcher(String` **`requestURI`**`)` bezieht. Zum Schluss braucht nur noch `RequestDispatcher.forward(HttpServletRequest, HttpServletResponse)` aufgerufen werden und die Bearbeitung des eingegangenen Requests ist abgeschlossen.

Bei jedem weiteren Request wiederholen sich Schritt eins und zwei von Szenario zwei.

1.5 Zusammenfassung

Der Webframework Struts ist das bekannteste und bis heute am häufigsten verwendete Framework für die Entwicklung von Webapplikationen. Es basiert auf dem Model View Controller Design Pattern mit den JSPs als Views, den Models als `ActionForm` Klassen und den `Actions` als Controllerklassen.

Die Funktionalität lässt sich in zwei große Bereiche aufteilen, zum einen das Hochfahren von Struts Webanwendungen und dem anderen das Request Processing. Angestoßen wird das Hochfahren vom `ActionServlet` in der Funktion des Frontcontrollers mit der Methode `ActionServlet.init()`, was als Kernfunktionalität, das Auslesen der Registry `struts-config.xml` beinhaltet, in der alle MVC Komponenten dem Framework bekannt gemacht werden. Das Request Processing beginnt mit `ActionServlet.service(HttpServletRequest, HttpServletResponse)`, wo es darum geht, die nun im Hauptspeicher befindlichen MVC Informationen der `struts-config.xml` zum Funktionieren der Anwendung, sprich dem Nutzen für die Benutzer, zu verwenden.

Mit diesen Eigenschaften wird das Ziel erreicht, dem Entwickler ein Programmiermodel zu bieten, mit dem er sich voll und ganz auf die Entwicklung der Anwendung konzentrieren kann, ohne sich um grundlegende Dinge kümmern zu müssen, die überhaupt erst die Webentwicklung ermöglichen.

Dies ermöglicht eine entscheidend leichtere Fehlersuche und Wartbarkeit auch bei sehr großen und komplexen Anwendungen.

2 Sun Java Server Faces

Die Java Server Faces sind die Antwort der Firma Sun Microsystems auf ASP.Net von Microsoft. Dies wird deutlich, wenn man bedenkt, dass Sun eine eigens auf JSF zugeschnittene Entwicklungsumgebung, das Java Studio Creator, herausgegeben hat, das es dem Entwickler erlaubt, wie das Visual Studio .Net für ASP.Net Weboberflächen per Drag and Drop zu erstellen. Etwas vergleichbares gibt es für Struts, zumindest bisher, nicht.

Ebenso wie Struts bietet JSF ein einheitliches standardisiertes Programmiermodel nach dem MVC Model. Die Einarbeitungszeit zur ersten Erstellung von kleineren Webapplikationen ist mit der bei Struts vergleichbar, da grundlegende Prinzipien auch bei JSF zu finden sind, siehe 2.1.

Es ist absehbar, dass JSF neben Struts eine ernsthafte Alternative zur Entwicklung von Webanwendungen wird. Dies wird plausibel, wenn man sich auch hier die Menge an Literatur an sieht, die z.B. bei den Verlagen O'Reilly und Addison-Wesley ansieht. Auch in Magazinen wie der Ausgabe September 2004 des Java Magazins wird dieser Trend deutlich.

Daher verfolgt das Kapitel zwei das gleiche Ziel wie Kapitel eins. Die Betrachtung von JSF hinter den Kulissen, sprich den Quelltext, was in der Literatur weniger häufig zu finden ist, um dem interessierten Entwickler einen tiefen Einblick zu geben, womit er es mit JSF zu tun hat. Damit der Leser den Ausführungen folgen kann, sind dieselben Vorkenntnisse wie bei Kapitel zwei erforderlich.

2.1 Kapitelübersicht

Die am häufigsten verwendete Alternative zum Webanwendungsentwicklungsframework Jakarta Struts sind die Sun Java Server Faces. Um dem Leser die Java Server Faces vorzustellen, werden in Kapitel 2.2 die grundlegenden individuellen Konzepte dieses Frameworks dargelegt, die es bei Struts nicht gibt oder die völlig anders realisiert sind. Ist diese Verständnisbasis geschaffen, folgt wie in Abschnitt 1.3 und 1.4 eine detaillierte Quelltextanalyse in Kapitel 2.3 und 2.4, um die genaue Funktionsweise von JSF zu verstehen. Eine Übersicht über die Realisierung des MVC Musters und die Unterschiede zwischen Struts und JSF, sind zum Einstieg sehr hilfreich, wenn nicht unverzichtbar.

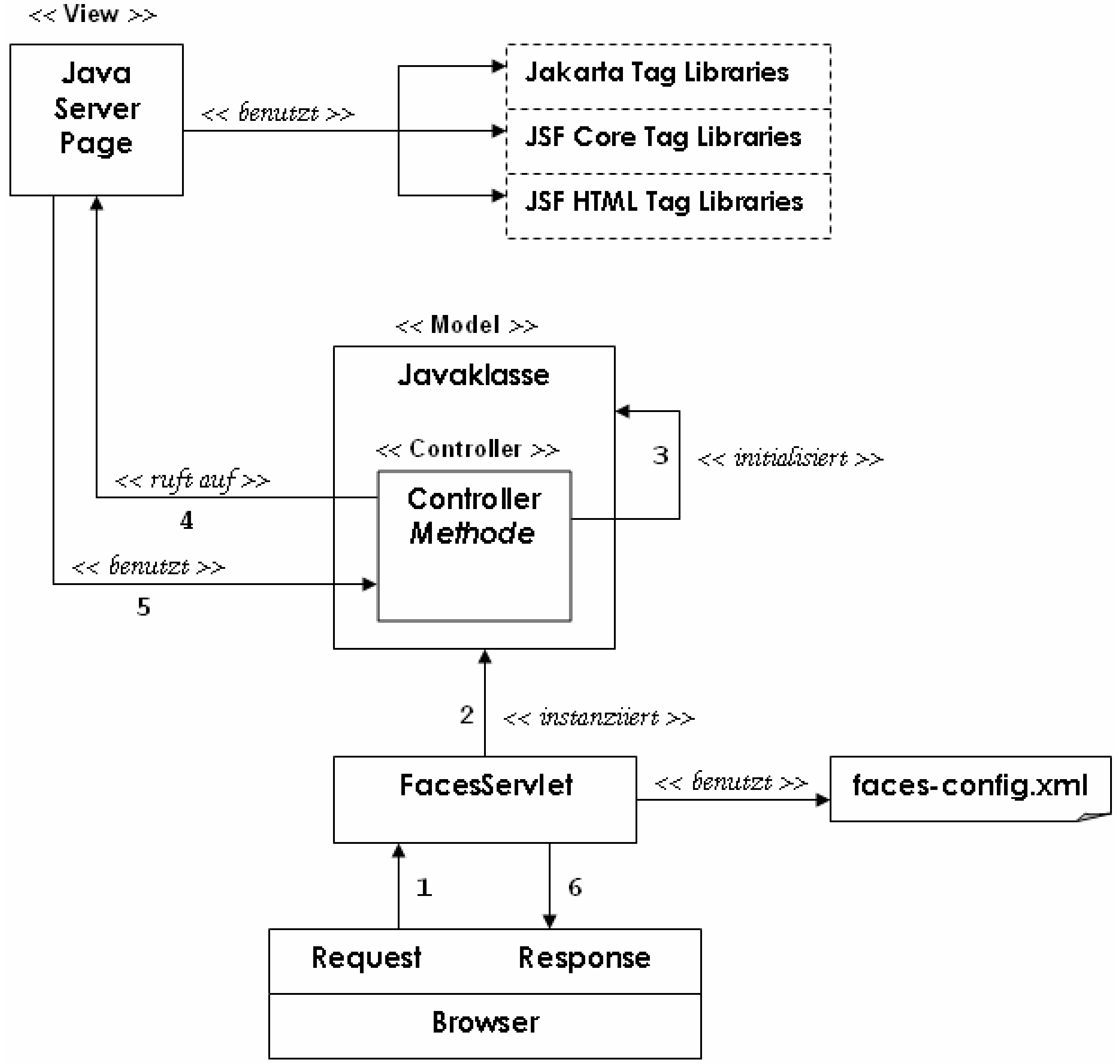


Abbildung 4: Umsetzung des MVC Modells in JSF.
Für Details siehe die folgenden Ausführungen

Der wichtigste Unterschied bei der Umsetzung des MVC Modells ist bei JSF die Zusammenführung von Model und Controller in eine Modelklasse, in der der Controller als Methode implementiert wird, mit der das Framework per String Rückgabewert und `faces-config.xml` das Navigationsverhalten zwischen den View Elementen steuert.

Die Java Server Faces Specification, siehe[3], erläutert die grundlegenden Konzepte wie den Request Processing Lifecycle, dem Value Binding und Method Binding, den Requestdaten als Status eines Java Server Faces, also einer JSP View, dem Komponenten und Rendering Modell , dem Validierungsunterframework und dem Lifecycle Management. Die Art der Themenbehandlung zielt wie bei [1] und [2] darauf ab, dem noch in JSF unerfahrenen Entwickler zu zeigen, wie er diese Konzepte nutzt und was er wissen und tun muss, um mit deren Hilfe seine Anwendung zu realisieren.

Das Oreilly Buch unter [4] ist das JSF Pendant zu [2]. Auch hier wird der Leser Schritt für Schritt zur selbständigen Erstellung einer JSF Anwendung herangeführt, die vom Umfang her denen ähnelt, die bei der JSF Distribution zu finden sind.

Die Konzepte des JSF Frameworks werden in den Abschnitten 2.3 und 2.4 dahingehend behandelt, wie sie als Javacode realisiert sind. Für den bereits fortgeschrittenen JSF Webentwickler sind diese Erkenntnisse von Belang, wenn er erfahren möchte, wie alternativ zu Struts ein Webentwicklungsframework funktioniert.

2.2 Grundstrukturen und Basiskomponenten

Eine grundlegende Eigenschaft des Java Server Faces Frameworks ist dessen Zweiteilung eine abstrakte Definition und Deklaration auf der einen, und dessen herstellerabhängige Implementierung auf der anderen Seite. Die Definition und Deklaration besteht überwiegend aus Interfaces und abstrakten Klassen, die definieren und vorschreiben, **was** zu implementieren ist; d.h., es steht fest, welche Methoden von welchen Klassen zu implementieren sind, damit der JSF Framework eines Herstellers funktioniert. Mit diesem Ansatz wird ausserdem erreicht, dass die Implementierung für den Entwickler gekapselt, bzw. transparent ist. Er muss sich also nicht darum kümmern, welchen JSF Framework Hersteller er benutzt, was im Idealfall bedeutet, dass er den Provider einfach durch einen anderen austauschen kann, ohne Änderungen und Anpassungen an seiner Applikation vornehmen zu müssen.

Dies wird dadurch erreicht, dass er jeweils nur die abstrakten Komponenten des Frameworks bei der Anwendungsprogrammierung benutzt. Hinter diesen Interfaces und abstrakten Klassen steht dann für den Entwickler unsichtbar die Implementierung des jeweiligen Herstellers.

Zu erkennen ist der abstrakte Teil von JSF daran, dass alle Klassen im Package `javax.faces` oder in deren Unterpackages liegen. Im Rahmen der folgenden Untersuchungen wird die Referenzimplementierung von Sun in der gegenwärtig aktuellen Version 1.1 verwendet. In der kleinen nachstehenden Tabelle sind Provider aufgelistet, die eine JSF Implementierung zur Verfügung stellen, oder JSF UI Komponenten, die sich in eine JSF Applikation einbinden lassen.

Name	Link
`MyFaces`	http://www.myfaces.org
`Smile`	http://smile.sourceforge.net/index.html
`ADF Faces Components`	http://www.oracle.com/technology/products/jdev/htdocs/partners/addins/index.html
`javascript4jsf`	http://javascript4jsf.dev.java.net/
`jsfcomp`	http://sourceforge.net/projects/jsfcomp/
`OurFaces`	http://www.ourvision.de/ourfaces/index.html
`WebGalileo Faces Components`	http://www.jscape.com/webgalileofaces/
`WebTree`	http://www.otrix.com/products/webtree/index.html
`WebMenu`	http://www.otrix.com/products/webmenu/index.html
`WebGrid`	http://www.otrix.com/products/webgrid/index.html

Tabelle 4: JSF Implementierungshersteller und freie JSF GUI Komponenten

2.2.1 Grundlegende Klassen der Java Server Faces

Es gibt Frameworkobjekte in denen Informationen hinterlegt sind, die sowohl für den Entwickler als auch für das Framework selbst für dessen Services von Bedeutung sind. Diese globalen Objekte werden ähnlich wie im Kapitels 2.2 hinsichtlich der Art von Information, die sie beinhalten, vorgestellt. Zu diesen Objekten gehören:

`javax.faces.context.FacesContext`

Für jeden einzelnen Request erzeugt das JSF Framework ein `FacesContext` Objekt. Über dieses Objekt sind diejenigen wichtigen globalen Objekte, der `ExternalContext`, das `Application` und das `UIViewRoot` Objekt, verfügbar. So ist der `FacesContext` der Zugang zu allen Request bezogenen Informationen.

`javax.faces.context.ExternalContext`

Der `ExternalContext` ist ein zentrales Zugriffsobjekt für alle Informationen rund um die Servlet API Objekte:

```
javax.servlet.ServletContext
javax.servlet.http.HttpServletRequest
javax.servlet.http.HttpServletResponse
javax.servlet.http.HttpSession
javax.servlet.Cookie
```

Der `ExternalContext` stellt die Infrastruktur in Form von eigens implementierten Map Objekten als Inner Classes, die von `java.util.AbstractMap` abgeleitet sind, zur Verfügung, um `Request`, `Session` und `Cookie` Objekte zu sichern. Auf die gleiche Weise werden Requestparameter und –werte, sowie Initparameter, die ursprünglich aus der `web.xml` stammen, hinterlegt.

Ebenfalls wird der gerade eingegangene `Request` und seine zugehörige `Session` als einfaches Attribut festgehalten.

`javax.faces.component.UIViewRoot`

Jede JSP Seite besteht aus einer Reihe von GUI Elementen. Eine JSF – JSP Seite nutzt die Elemente, die in der Tag Library Definition Datei `html_basic.tld` vorzufinden sind. Für jedes HTML GUI Element, wie z.B. das Textfeld oder die Combobox steht ein Tag zur Verfügung, der in dieser `tld` Datei definiert ist. Der jeweilige Tag beinhaltet neben dem Tag Namen den vollklassifizierenden Klassennamen der Implementierungsklasse für die GUI Komponente. Für das Textfeld sieht dies folgendermaßen aus:

```
<tag>
  <name>inputText</name>
  <tag-class>com.sun.faces.taglib.html_basic.InputTextTag</tag-class>
  <tei-class>com.sun.faces.taglib.FacesTagExtraInfo</tei-class>
  <body-content>JSP</body-content>
  <description>...</description>
  <attribute>
    <name>id</name>
    <required>false</required>
    ...
  </attribute>
  ...
  <attribute>
    <name>value</name>
    <required>false</required>
    ...
  </attribute>
```

```
  ...
</tag>
```

Das `UIViewRoot` Element ist nach der JavaServer™ Faces Specification auf jeder JSF - JSP Seite das Wurzelelement für die GUI Komponenten Tags. Es entsteht also ein Baumhierarchie von Elementen, da auch die Kind GUI Elemente von `UIViewRoot` wiederum eigene Kindelemente besitzen können.

Für `UIViewRoot` existiert kein Eintrag in der `html_basic.tld`. Es besitzt demnach auch kein graphisches Erschneinungsbild wie die anderen GUI Komponenten. Da es aber trotzdem ein GUI Element ist, muss es in einer `tld` Datei definiert sein. Dies erfolgt in der zweiten `tld` Datei, die auf jeder JSP eingebunden wird: `jsf_core.tld`. Ähnlich wie die Tag Deklaration des Textfeld GUI Elements sieht die Deklaration für `UIViewRoot` aus:

```
<tag>
  <name>view</name>
  <tag-class>com.sun.faces.taglib.jsf_core.ViewTag</tag-class>
  <tei-class>com.sun.faces.taglib.FacesTagExtraInfo</tei-class>
  ...
</tag>
```

Seine Funktion als Wurzelelement auf einer JSP Seite wird von mehreren JSF Framework Klassen verwendet. Diese Funktionen drehen sich um das Verarbeiten von Daten, die in Verbindung mit den einzelnen GUI Komponenten stehen. Als Wurzelelement besitzt `UIViewRoot` Zugang zu allen seinen Kindelementen, und damit zum gesamten Inhalt einer JSF JSP.

`javax.faces.webapp.FacesServlet`

Wie das `ActionServlet` beim Struts Framework, ist das `FacesServlet` für JSF die Schaltzentrale für die Initialisierung des Frameworks und die Requestbearbeitung. Es gibt jedoch einen entscheidenden Unterschied zwischen dem `ActionServlet` und dem `FacesServlet`. Das `FacesServlet` ist als `final` implementiert, was für den Entwickler bedeutet, dass er keine zusätzliche Funktionalität durch Ableitung hinzufügen kann, wie es bei Struts möglich ist.

Wie beim `ActionServlet` erfolgt die Initialisierung in der `FacesServlet.init(ServletConfig)` Methode, und die Requestbearbeitung in der `FacesServlet.service(ServletRequest, ServletResponse)` Methode.

`javax.faces.el.ValueBindung`

Beide, das `ValueBinding` und das `MethodBinding`, haben gemeinsam, dass sie einen value binding Ausdruck repräsentieren, der mit einer Ausnahme der Syntax der Expression Language der JSP 2.0 Spezifikation genügen muss. Ein EL Ausdruck hat folgendes Format:

```
#{<Ausdruck>}
```

Der Beginn eines Ausdruck bilden die Zeichen `#{`. Beendet wird er mit dem Zeichen `}`. Das `#` ersetzt das `$` der JSP 2.0 Spezifikation. Für den Inhalt der Expression gibt es mehrere Syntaxregeln, also mehrere Varianten und Spezialfälle, die für spezielle Anwendungsfälle wichtig sind. Im Rahmen dieser Erläuterungen ist die Betrachtung der value binding Expressions auf den häufigsten Anwendungsfall beschränkt. Für das gesamte Spektrum der EL Syntax siehe die Kapitel 2.3 bis 2.9 der JavaServer™ Pages Specification 2.0. Die häufigste Verwendung von value binding Ausdrücken sieht folgendermaßen aus:

```
#{name1.….nameN}
```

Der Ausdruck besteht innerhalb der Begrenzungszeichen aus einem oder mehreren alphanumerischen Bezeichnern, die voneinander durch einen Punkt getrennt sind. Die Bezeichner stimmen entweder Attributnamen von Javaobjekten überein, oder müssen in der `faces-config.xml` als Wert eines Elements auffindbar sein. Ein Beispiel ist der Wert eines `<managed-`

bean-name> Elements, das zu einem <managed-bean> Element gehört.

Zwei klassische Beispiele, die den häufigsten Anwendungsfall abdecken sind die Benutzung des outputText und inputText Elements aus der html_basic.tld in einer JSF JSP. Ein value binding Ausdruck, der als Attribut zu diesen beiden Tags gehört, wird folgendermaßen benutzt:

```
<h:inputText  value="#{beanName1.attribute1}"/>, bzw.
<h:outputText value="#{beanName2.attribute2.attribute21}"/>
```

Die Auswertung dieser beiden der Art nach identischen value binding Ausdrücke besteht aus zwei Teilen, die von zwei Klassen durchgeführt wird. Für den ersten Teil der Auswertung ist der VariableResolver zuständig, der immer den ersten Bezeichner vor dem ersten Punkt bearbeitet. Für den Rest des Ausdrucks gibt es den PropertyResolver.

javax.faces.el.MethodBinding

Aufgabe des MethodBinding ist es, anhand eines method binding Ausdrucks eine Methode eines zuvor vom VariableResolver erzeugten Javaobjekts aufzurufen. Die method binding Ausdrücke haben genau dieselbe Syntax und unterscheiden sich daher nicht den value bindings. Bei den Attributen der folgenden Tag Repräsentationen von HTML UI Elementen kommen MethodBindings am häufigsten vor:

```
<h:commandButton actionListener="#{beanName.actionListenerMethod}" …/>
<h:commandButton action="#{beanName.method}" …/>
<h:inputText validator="#{beanName.validatorMethod}" …/>
<h:selectOneMenu valueChangeListener="#{beanName.changeListenerMethod}"/>
```

Die Syntax bei den method bindings aller vier Tagattribute ist dieselbe. Es ist jeweils ein Javaobjektname anzugeben und durch einen Punkt getrennt, dessen gewünschte Methode. Jedoch erfordert das JSF Framework jeweils unterschiedliche Methodenprototypen. D.h., dass die Methode einen Rückgabetyp haben muss oder nicht, und verscheidene oder auch keine Methodenparameter erforderlich sind.

Das Framework untersucht erstens den angegebenen Namen für das Javaobjekt, ob es in der faces-config.xml als Inhalt eines <managed-bean-name> Element, das zu einem <managed-bean> Element gehört, vorhanden ist. Als zweites wird überprüft, ob das existierende Javaobjekt eine Methode hat, die mit dem Namen aus dem jeweiligen Tagattribut, dem Namen nach dem Punkt im method binding Ausdruck, übereinstimmt. Ist auch dies der Fall, muss die so identifizierte Methode noch den richtigen oder keinen Rückgabewert haben und die richtige Parameterliste aufweisen.

javax.faces.application.Application

Ein Application Objekt ist eine Zentrale für Singleton Objekte, die für die gesamte JSF Applikation von Bedeutung sind. Auch die Application selbst ist von anwendungsglobaler Sichtbarkeit und muss threadsafe implementiert werden. Zu den Objekten, die Framework weite Funktionalität zur Verfügung stellen, gehören:

- javax.faces.event.ActionListener
  Die ActionListener Implementierung des JSF Frameworks ist der Startpunkt der Bearbeitung einer jeden vom Benutzer ausgelösten Aktion. Diese Aktion können gegenwärtig von zwei HTML GUI Komponenten ausgelöst werden: dem HtmlCommandButton und dem HtmlCommandLink. Beide erben als einzige GUI Komponenten von javax.faces. component.UICommand. In seiner Callback Methode ActionListener.process Action(ActionEvent) passieren drei wesentliche Dinge:

  1. Ermitteln des MethodBinding zur Ermittlung der Methode, die durch das Betätigen von HtmlCommandButton oder HtmlCommandLink aufgerufen werden soll. Dadruch erfolgt die Ausführung der vom Entwickler implementierten Applikationslogik.

2. Ist die Applikationslogik abgearbeitet muss auf die nächste JSP verzweigt werden. Dazu benutzt der `ActionListener` den `NavigationHandler`. Der `NavigationHandler` besitzt die Callbackmethode `NavigationHandler.handleNavigation(FacesContext, String source, String target)`. Diese ermittelt mittels der beiden `String` Parameter die richtige Seite auf die als nächstes aufzurufen ist.

3. Die letzte Aktion des `ActionListener`s ist das Anstoßen der letzten Phase des Request Processing Lifcycle, der `RenderResponse` Phase.

Der `ActionListener` bei den JavaServer Faces ist nicht ganz mit seinem Swing Pendant, dem `java.awt.ActionListener` zu verwechseln. Bei JSF gibt es nur einen einzigen, der wiederum nur über eine einzige Quelle dem `Application` Objekt zu beziehen ist. Bei Swing kann es beliebig viele geben, die der Entwickler stets selbst zu implementieren hat. Eine JSF Implementierung stellt immer einen Default `ActionListener` bereit, sodass der Web Entwickler nur die Konventionen seiner Benutzung beachten muss.

- `javax.faces.application.NavigationHandler`
  Die Kernfunktionalität des `NavigationHandler` wurde bereits beim `ActionListener` angesprochen. Das Anzeigen der nächsten View Komponente nach einer Benutzerinteraktion unter Berücksichtigung des Rückgabewerts der Aktionsbearbeitungsmethode.

  Die hierfür zuständige Callbackmethode `NavigationHandler.handleNavigation(FacesContext, String source, String target)` erledigt dies unter Zuhilfenahme der beiden String Parameter und der Inhalte der `faces-config.xml`. Es muss hierzu ein `<navigation-rule>` Element vorliegen dessen Kindelement `<from-view-id>` den Wert von `String source` haben muss. Zweite Bedingung für ein erfolgreiches Verzweigen auf die richtige Seite, ist ein `<navigation-case>` Element, dass ein `<from-outcome>` Element besitzt, dessen Wert `String target` entspricht. Zur Verdeutlichung ein Beispielausschnitt aus einer `faces-config.xml`:

```
<navigation-rule>
  <from-view-id>/view/module/lectureSearch.jsp</from-view-id>
  <navigation-case>
    <from-outcome>success</from-outcome>
    <to-view-id>/view/module/myModules.jsp</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>subscribe</from-outcome>
   <to-view-id>/view/module/subscribe.jsp</to-view-id>
  </navigation-case>
</navigation-rule>

<managed-bean>
  <managed-bean-name>lectureBean</managed-bean-name>
  <managed-bean-class>com.test.LectureBean</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
</managed-bean>
```

Auf der Viewseite `/view/module/lectureSearch.jsp` wird durch einen submit Button `<h:commandButton action="#{lectureBean.actionMethod}"/>` eine Benutzerinteraktion ausgelöst.

Das Beanobjekt `lectureBean` ist in der `faces-config.xml` eindeutig definiert und muss eine Methode `public String com.test.LectureBean.actionMethod()`

besitzen. Ist der Rückgabewert dieser Methode `success`, wird auf die im `navigation-case/to-view-id` Element hinterlegte Seite `/view/module/myModules.jsp` verzweigt.

Das Aufbauen der GUI Elementbaumstruktur umfasst das Erzeugen eines `UIViewRoot` Objektes. Für den Aufbau dieses Objekts stehen dem `StateManager` folgende Methoden zur Verfügung. Die Einrückungen spiegeln die Aufrufhierarchie wider.

- ```
  StateManager.restoreView(...)
     StateManager.restoreTreeStructure(...)
        StateManager.restoreComponentTreeStructure(...)
     StateManager.restoreComponentState(...)
  ```
- ```
  StateManager.saveSerializedView(...)
     StateManager.getTreeStructureToSave(...)
        StateManager.buildTreeStructureToSave(...)
     StateManager.getComponentStateToSave(...)
  ```

Die Werkzeuge dieser Methoden zur Erstellung der Baumstruktur ist zum einen das Objekt `com.sun.faces.util.TreeStructure` als Wrapperobjekt für ein `UIComponent` Objekt, und die Rekursion zum Traversieren und Erstellen der Baumhierarchie von `TreeStructure` Wrapperobjekten. Die beiden Methoden `StateManager.restoreView(...)` und `State Manager.saveSerializedView(...)` der obersten Hierachie werden von den Phasen des Request Processing Lifecycle verwendet.

- `javax.faces.application.ViewHandler`
  Die Aufgabe des `ViewHandler` ist, View Seiten dem Framework erreichbar zu machen. Um diese Aufgabe zu erfüllen, besitzt er u.a. drei wichtige Methoden, die nachfolgend kurz erläutert werden.

  `void ViewHandler.renderView(FacesContext, UIViewRoot)`
  Das Rendern bei den Java Server Faces umfasst die Untersuchung des Requesttyps und des Mappings des eingehenden Requests. Dazu wird der request Uniform Resource Identifier, also das `viewId` Attribut überprüft, ob es das Format eines JSF Request hat. Ist dem so, kann der Request über den `ExternalContext` vom `javax.servlet.Request Dispatcher` ausgeführt und weitergeleitet werden.

  Ist dem nicht so, muss die `viewId` modifiziert werden. Damit ist ein weiterer Untersuchungsgegenstand das Mapping, also die Frage ob der eingehende Request vom `FacesServlet` kommt, oder eines anderen Ursprungs ist. Jeder nicht `FacesServlet` Request muss in das entsprechende Format überführt werden. Das bedeutet, dass der request URI mit dem Mapping versehen werden muss, das in der Standardkonfiguration einer JSF Applikation, in der `web.xml` unter dem Element `servlet-mapping/url-pattern` definiert ist.

  `UIViewRoot ViewHandler.restoreView(FacesContext, String viewId)`
  Diese Methode übernimmt die Wiederherstellung eines `UIViewRoot` Objektes. Bevor dieses Objekt bereitsteht, müssen folgende Arbeitsschritte erledigt werden:

  1. Das Setzen des Character Encodings, z.B. `UTF-8` und des Content Types wie z.B. `text/plain` oder `text/html`.
  2. Das Ermitteln der `viewId`, also des request URIs unter Berücksichtigung des in der `web.xml` definierten Mappings, ähnlich der `ViewHandler.renderView()` Methode.
     a. `viewId` konnte **nicht** ermittelt werden: Weitergabe der Requestbearbeitung an den `ExternalContext`.
     b. `viewId` konnte ermittelt werden: Bezug des `UIViewRoot` Objekts vom

`StateManager` und dessen `restoreView()` Methode.

`UIViewRoot ViewHandler.createView(FacesContext, String viewId)`

Erstellt ein neues `UIViewRoot` Objekt und setzt dessen wichtigste Attribute. Dazu gehören:

- die `viewId` stellvertretend für den relativen Contextpfad einer jsp. Z.B. `/view/index.jsp`, wobei das Verzeichnis `/view` direkt im Webapplikations-hauptverzeichnis liegt.
- Das Bereitstellen eines `java.util.Locale` Objektes, unter Berücksichtigung der Einträge in der `faces-config.xml`. Falls keine Konfiguration vorhanden, oder keine `Locale` von der View Seite vor diesem Request, muss die `ViewHandler` Implementierung dem `UIViewRoot` Objekt ein Standart `Locale` Objekt mitgeben.
- Das Setzen einer `renderKitId` entweder über das `javax.faces.application.Application` Objekt oder über die Standart id von `javax.faces.render.RenderKitFactory` mit deren Konstante `HTML_BASIC_RENDER_KIT`.

Die folgenden Unterpunkte behandeln JSF spezifische Konzepts. Deren Funktionsumfang wird je einführend dargestellt, da sie für das Verständnis der Ausführungen in den Kapiteln 2.3 und 2.4 notwendig sind.

2.2.2 Der Request Processing Lifecycle

Ein grundlegendes Merkmal von webbasierten Anwendungen ist, dass GUI Komponenten aufgrund der Arbeitsspeicherverwaltung nicht im Arbeitsspeicher gehalten werden können, wie dies bei Desktopanwendungen der Fall ist. D.h., wenn für jeden Request jede Komponente auf jeder Seite gespeichert würde, wäre eine schnelle Überlastung des Servers die Folge.

Daraus folgt, dass ein anderes Konzept realisiert werden muss, um die vom Benutzer ausgelösten Ereignisse zu bearbeiten und die Resultate in der Response an den richtigen Client zurückzusenden. Ebenfalls kann dem Server keine Benachrichtigung gesendet werden, wenn sich auf der Clientseite ein Wert einer Komponente auf einem View ändert bevor ein Request gesendet wurde. Bei stand alone Anwendungen ist dies ganz einfach per Notification machbar.

Ein Beispiel ist die Auswahl eines anderen Elements aus einer Listbox. In einer Standalone-anwendung gibt es einen der Listbox GUI - Komponente maßgeschneiderten ValueChange Listener, der eine Callbackmethode implementiert, die automatisch aufgerufen wird. Hier ist dann die Applikationslogik, die auf dieses Ereignis abgearbeitet werden soll, ganz einfach abrufbar.

Da diese Notification in einer Webanwendung nicht existiert, wurde für JSF ein anderer Weg gewählt, der zwar ebenfalls zuverlässig funktionert, jedoch einen Nachteil hat. Der gewählte Weg wird über clientseitiges Scripting mit JavaScript implementiert, die aber, wenn beim Browser aus Sicherheitsgründen kein JavaScript aktiviert ist, wirkungslos bleibt.

Neben der Benachrichtigung und Bearbeitung von vom Benutzer ausgelösten Ereignissen müssen in der nach dem Request folgenden Reponse alle GUI Elemente der View Seite wieder zur Verfügung stehen, bzw. wieder aufgebaut werden. Da die GUI Elemente nicht gespeichert werden, müssen die richtigen Daten, die den Wiederaufbau der Response Seite ermöglichen irgendwie gespeichert und anschließend durch eine weiteres Konzept bearbeitet werden, bis zur Anzeige der vom Client angeforderten Seite. Zu diesem Zweck gibt es den Request Processing Lifecycle.

Der Request processing lifecycle besteht aus sechs Phasen.

```
RestoreViewPhase
   ApplyRequestValuesPhase
      ProcessValidationsPhase
         UpdateUserModelPhase
```

```
InvokeApplicationPhase
  RenderResponsePhase
```

Jeder Phase ist eine Implementierungsklasse zugeordnet, die alle von der Basisklasse `com.sun.faces.lifecycle.Phase` erben und die Methoden `Phase.execute(FacesContext)` und `Phase.getId()` implementieren. Alle Implementierungsklassen befinden sich im selben Package `com.sun.faces.lifecycle`. Die `Phase.getId()` Methode liefert ein `PhaseId` Objekt zurück, das die jeweilige Phase als eine der sechs Phasen identifiziert.

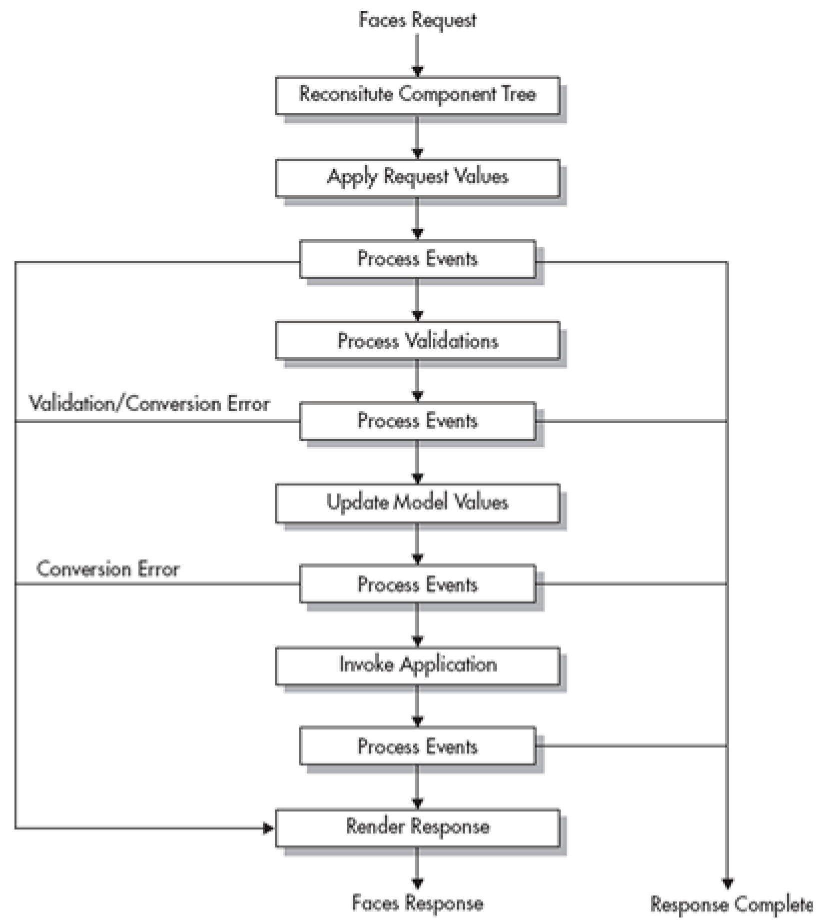


Abbildung 5: Der JSF Request Processing Lifecycle.
Grundlegende Informationen zu jeder Phase werden direkt nachfolgend gegeben

In der `RestoreView.execute(FacesContext)` Methode der **`RestoreView`** Phase wird zuerst geprüft, ob der `FacesContext` bereits ein `UIViewRoot` Objekt besitzt. Ist dies der Fall, bedeutet dies, dass dieses Objekt schon seit dem letzten Request existiert und für diesen immer noch aktuell ist. Wurde beispielsweise für diesen Request nur der Zustand einer GUI Komponente, wie einer Listbox geändert, sprich, dass ein anderes Listenelement ausgewählt wurde, wurde ein neuer Request an den Server gesendet, ohne dass auf eine andere Seite verzweigt werden müsste. Demzufolge muss auch kein neues `UIViewRoot` Objekt erzeugt werden, das eine neue GUI Element Hierarchie repräsentiert.

Im nächsten Schritt wird `UIViewRoot` das `Locale` Objekt übergeben, das aus dem `ExternalContext` des übergebenen `FacesContext` bezogen wird. Somit ist klar, welches Sprach `.properties` File herangezogen werden muss, damit die Labels, die durch das `<h:outputText>` Element abgebildet werden, den Text in der richtigen Sprache widergeben. Zu guter Letzt müssen die `valueBinding` Attribute aller Komponenten, des `UIViewRoot` GUI

Wurzelelements gesetzt werden. Dies ist notwendig, um in späteren Lifecyclephasen die entsprechenden Komponenten an die Attribute des Beanobjektes zu binden. Auf diese Weise ist garantiert, dass diejenigen UI Elemente, die ein binding Attribut besitzen, dem Benutzer immer die Daten des richtigen Beanfeldes anzeigen. Für den Fall, dass der `FacesContext` des eingegangenen Requests das immer noch aktuelle `UIViewRoot` Objekt des vorangegangenen Requests besitzt, ist die erste Phase des Request Processing Lifecycle beendet.

In der Mehrzahl der Fälle muss das `UIViewRoot` Objekt neu erzeugt werden, denn sobald auf eine neue oder andere Seite verzweigt werden muss, ist erstens der Servletpfad in der Aderssleiste des Browsers ein anderer und zweitens hat die Seite dieses Pfades einen gänzlich anderen Inhalt. Der Servletpfad, mit dessen Wert das `viewId` Attribut des `UIViewRoot` Objekts initialisiert wird, muss zuerst ermittelt werden. Je nachdem, ob es sich beim Servletmapping um Postfix- oder Prefixmapping handelt, müssen für die `viewId` andere Informationsquellen abgefragt werden.

Die JSF Referenzimplementierung unternimmt zwei Anläufe, um den richtigen Wert für das `viewId` Attribut des `UIViewRoot` Objekts zu ermitteln. Zuerst versucht es die `RestoreView` Phase mit der Servlet API Request Konstanten `"javax.servlet.include.path_info"`. Liefert dieser Versuch kein brauchbares Ergebnis wird über die zweite Standartkonstante `"javax.servlet.include.servlet_path"` versucht, eine `viewId` eine zu erhalten. Der Servlet Path entspricht dem Wert der in der `web.xml` unter dem Element `web-app/servlet-mapping/url-pattern` steht. Dies ist derjenige relative Pfadanteil unter dem ein Serlvet im Browser aufgerufen wird. Z.B.: Sei das servlet mapping `"init"` und das Webapplikationshauptverzeichnis `"test"`, so wird das Servlet mit der Adresse `http://localhost:8080/test/init` angesprochen. Kann auch hier keine Resource ermittelt werden, bricht das JSF Framework den gesamten Request Processing Liefcycle ab, und beendet den Request mit einer `FacesException`.

Wurde auf einem der beiden möglichen Wege ein Wert für das `viewId` Attribut gefunden, gibt es wiederum zwei Quellen zum Erstellen eines vom `viewId` Attribut abhängigen `UIView Root` Objekts. Zuerst wird die weniger aufwendige Möglichkeit über den `ViewHandler` versucht. Dieser verfügt über einen Cache von `UIViewRoot` Objekten. Liegt in diesem Cache ein der `viewId` passendes `UIViewRoot` vor, kann dieses mit dem `ViewHandler` und dessen Methode `ViewHandler.restoreView(FacesContext, viewId)` wiederhergestellt werden, was im Gegensatz zur Neuerstellung deutliche Performancevorteile bringt.

Lässt sich auf diesem Wege kein `UIViewRoot` Objekt herstellen, muss die `ViewHandler.createView(FacesContext, viewId)` Methode bemüht werden, die Funktionalitäten ausführt, die in den Erläuterungen zum `ViewHandler` beschrieben sind. Das nun auf jeden Fall vorhandene `UIViewRoot` Objekt muss nun noch dem `FacesContext` bekannt gemacht werden, und dessen Kindelemente auf `binding` Tagattribute untersucht werden, analog zu dem ersten erläuterten Fall, wenn das `UIViewRoot` Objekt bereits im `FacesContext` des vorangegangenen Requests vorliegt.

Die **`ApplyRequestValues`** Phase hat die Aufgabe des Updatens der GUI Komponenten auf den aktuellen Stand unter Berücksichtigung von Requestparametern, z.B. nach dem Klicken auf einen `CommandButton`, `CommandLink` oder einer ListBox beim Auswählen eines anderen Listenelements. Das Updaten erfolgt in `ApplyRequestValuesPhase.execute(Faces Context)` über den Aufruf der `UIComponent.processDecodes(FacesContext)` Methode. Alles beginnt mit `UIViewRoot` und von dort aus rekursiv für alle Kind- und Kindeskindelemente.

In der `processDecodes(FacesContext)` Methode wird jür jedes `UIComponent` Objekt und dessen Kinder die `UIComponent.decode(FacesContext context)` Methode aufgerufen und überprüft, ob das zu untersuchende `UIComponent` Objekt vom Typ `UIInput` ist. Ist dies nicht der Fall, bedarf es keiner weiteren Bearbeitung. Handelt es sich um eine von `UIInput` abgelei-

tete Komponente, z.B. ein Textfeld `InputText`, müssen die vom Benutzer eingegebenen Daten gesichert werden. D.h. einfach, dass die Eingabe in einem `Object` Attribut von `UIInput` hinterlegt wird.

Ist die `UIViewRoot.processDecodes(FacesContext)` Methode abgearbeitet, ist noch zu beachten, dass einige wenige GUI Komponenten Ereignisse ausgelöst haben können. Z.B. ein `ValueChangeEvent` einer Listbox oder einer Combobox, oder auch ein `ActionEvent` von einem `CommandButton`. Das Bearbeiten dieser Ereignisse nach vorherigem Feuern muss durch `UIViewRoot.broadcastEvents(FacesContext, PhaseId)` angestoßen werden. Was hier genau vor sich geht, behandelt Kapitel 2.2.4.

Zu berücksichtigen sind spezielle Typen von `UIComponents`. Diese erkennt man an der Implementierung von besonderen Interfaces wie `ActionSource`, `StateHolder`, `Value Holder`, und `NamingContainer` aus dem Package `javax.faces.component`. Die markanten Eigenschaften der Komponenten, die eines oder mehrerer dieser Schnittstellen implementieren, werden nachfolgend kurz vorgestellt.

- `ActionSource`
  Über die Basisklasse `UICommand` sind `HtmlCommandButton` und `HtmlCommandLink` `ActionSource` Derivate. Wird auf eine solche Komponente geklickt, wird eine Aktion angestoßen. Analog zu einem HTML submit Button ist ein `HtmlCommandButton` und `HtmlCommandLink` immer am Ende eines Form Tags, bzw. als Kindelement einer JSF Form GUI Komponente zu finden, mit der nach dem Absenden alle Werte der Form in der Aktionsbearbeitungsmethode der zuständigen Beanklasse weiterverarbeitet werden können.
- `StateHolder`
  Es gibt eine große Vielzahl von JSF GUI Komponenten, die zugleich eine `StateHolder` Komponente sind. Dazu gehören alle Ableitungen von `UIComponent`, sowie einzelne von `javax.faces.validator.Validator` und `javax.faces.convert.Converter` abgeleitete Klassen. All diese `StateHolder` implementieren u.a. zwei wichtige Methoden: `StateHolder.restoreState(FacesContext,Object state)` und `StateHolder.save State(FacesContext)`. Inhalt und Struktur des State `Object` wurde in den Ausführungen des `StateManager` beschrieben.
- `ValueHolder`
  Es gibt in der JSF GUI Komponentensammlung zwei Komponententypen, die einen `ValueHolder` bzw. einen `EditableValueHolder` darstellen. `UIInput` und `UIOutput`. Ihnen ist gemeinsam, dass sie ein Modelattribut vertreten. Damit stellen sie eine Verbindung zum Model – tier dar, sprich mit den Methoden `ValueHolder.getValue()` und `ValueHolder.setValue(Object)` können die Daten- und die Präsentationsschicht miteinander synchronisiert werden. Die Verbindung dieser beiden Schicht stellen value binding Ausdrücke her, die von den JSF Tag Implementierungsklassen in Verbindung mit den zwei erwähnten Methoden genutzt werden. Für jede GUI Komponentenklasse gibt es eine entsprechende Tag Komponentenklasse. Für das Textfeld ist die GUI Komponentenklasse `InputText`, die Komponentenklasse `InputTextTag`.

  Beispiele für `UIOutput` Komponenten sind `HtmlOutputText` und `HtmlOutputLink` und deren JSF Tagrepräsentationen. `UIInput` Elemente implementieren das spezielle `Value Holder` Interface `EditableValueHolder`. Dazu gehören u.a.: hidden buttons, Passworttextfelder, Textareas, Listboxen, Comboboxen, Textfelder und Radiobuttons. Über die `getValue()` und `setValue(Object)` Methoden hinaus, kann diesen GUI Komponenten mit Hilfe des `EditableValueHolder` noch `ValueChangeListener`s und `Validator`s über

method bindings zugeordnet werden, die auf Methoden von, der JSP, zugeordneten Modelklassen zeigen.

- NamingContainer
  NamingContainer Komponenten wie HtmlForm und HtmlDataTable sind Behälter für andere GUI Elemente wie UIInput und UIOutput Derivate. HtmlForm entspricht dem HTML <form> Tag, HtmlDataTable einem <table> Tag, in dessen Zeilen und Spalten Daten eines Beanobjektes geschrieben werden, das über einen value binding Ausdruck dem übergeordneten NamingContainer Element bekannt gemacht wurde.

In der **ProcessValidations** Phase werden alle GUI Komponenten einer JSF JSP auf die Deklaration eines Validators untersucht. Ein Validator ist für die Untersuchung für Benutzereingabedaten zuständig, um im Falle fehlerhafter Eingaben sinnvolle Fehler- oder Warnmeldungen sofort nach der Überprüfung anzuzeigen. Besitzt eine JSF UI Komponente, deren Implementierungsklasse das Interface EditableValueHolder implementieren muss, das Attribut validator, wie <inputText id="textfield" **validator="bean.validatorMethod"**> weiss das JSF Framework, dass der Wert der Eingabe, einer Methode der validatorMethod Methode einer bekannten Beanklasse übergeben werden muss, für die die folgende Signuatur vorgeschrieben ist:

```
void <valiatorMethodName>(FacesContext, UIComponent, Object)
```

Der Object Parameter ist der Wert, den es zu validieren gilt. UIComponent ist die GUI Komponente vom Typ EditableValueHolder. Ausgangspunkt der Validierungsphase ist die Methode ProcessValidationsPhase.execute(FacesContext). Für UIViewRoot wird process Validators(FacesContext) ausgeführt, und damit mittels Rekursion dieselbe Methode für jede EditableValueHolder GUI Komponente. UIInput beispielsweise fährt dann folgendermaßen fort: executeValidate(FacesContext) → validate(FacesContext) → validate Value(FacesContext, Object value). In letztere Methode erfolgt dann der eigentliche Validierungsprozess.

Der Validierungsprozess besteht darin, jedem dieser UI Komponente zugewiesenen Validatoren den Object value Parameter mit Validator.validate(FacesContext, UIComponent, Object) zu übergeben. Diese Methode beinhaltet dann die eigentliche Validierungslogik des Entwicklers.

In der **UpdateModel** Phase wird vorrausgesetzt, dass alle Daten, die über diesen Request versendet wurden, syntaktisch auf Korrektheit überprüft wurden. Unter Daten werden hier zum einen die Nutzdaten im hierarchischen GUI Kompnentenbaum (UIViewRoot) verstanden, die durch Benutzerinteraktion und durch die Bearbeitung von Applikationslogik versendet werden. Zum anderen sind dies requestbezogene Informationen wie z.B. der URL von Quell- und Ziel Resourcen, bzw. Seite, und andere Metadaten, die ein Requestobjekt begleiten.

Unter diesen Vorrausetzungen kann sich die UpdateModel Phase ihrer Hauptaufgabe widmen. Diese besteht darin, die UIViewRoot.processUpdates(FacesContext) Methode aufzurufen. Dies führt analog zur ApplyRequestValues Phase mit UIViewRoot.process Decodes(FacesContext) und ProcessValidations Phase mit UIViewRoot.process Validators(FacesContext) zum rekursiven Aufruf der processUpdates(FacesContext) Methode aller UIComponent GUI Komponenten.

Bei denjenigen UIComponents, die processUpdates(FacesContext) Methode implemen-

tieren, wie alle `UIInput` Komponenten, wird eine Methode `updateModel(FacesContext)` aufgerufen. Diese besorgt sich das `ValueBinding` Objekt der `UIInput` Tagrepräsentation `<inputText value="bean.attribute">` und aktualisiert es mittels `ValueBinding.setValue(FacesContext, value)` mit dem Wert `"bean.attribute"` des value Attributs der jeweiligen `UIComponent` Tagrepräsentation.

Die Implementierung der **`InvokeApplication`** Phase ist genauso schlank wie die der vorherigen Phasen mit nur einem nenneswerten Aufruf: `UIViewRoot.processApplication(FacesContext)`. In dieser Methode wiederum geschieht nur eines: Das Anstoßen des Feuerns von Ergeignissen, die sich auf das `Application` Objekt beziehen, gekennzeichnet durch eine entsprechende `PhaseId`. Die Aufgabe des Feuerns von Events egal für welchen Typ, bzw. egal für welche Request Processing Lifecycle Phase übernimmt eine einzige Methode: `UIViewRoot.broadcastEvents(FacesContext, PhaseId)`. Die genauen Mechanismen des Event handling wird im Kapitel 2.2.4 betrachtet.

Die **`RenderResponse`** Phase ist der letzte Schritt des Request Processing Lifecycle und für den Aufbau der nächsten oder den Wiederaufbau der gegenwärtigen Seite verantwortlich. Um dies anzustoßen, wird der `ViewHandler` über das `Application` Objekt geholt und dessen Methode `ViewHandler.renderView(FacesContext, UIViewRoot)` aufgerufen.

2.2.3 Das GUI Komponentenkonzept und Rendering

Das GUI Komponentenkonzept der Java Server Faces Spezifikation befindet sich im Paket `javax.faces.component`. Dort befinden sich Implementierungsklassen für alle gängigen HTML GUI Elemente, sowie diejenigen Interfaces `ActionSource`, `StateHolder`, `ValueHolder`, und `NamingContainer`, die einer GUI Komponente spezielle Eigenschaften verleihen. Mit diesem Komponentenset ist, von der reinen HTML Oberflächensicht aus betrachtet, ein herstellerunabhängiges Viewdesign der JSPs möglich.

Sind komplexere oder spezielle GUI Komponenten für JSF JSP Views für eine Webanwendung erforderlich, muss sich der Entwickler entweder selbst um die Implementierung kümmern oder er muss einen passenden Hersteller einer JSF Implementierung auswählen. Um anderen Herstellern die Erweiterung des HTML GUI Komponentenset der Java Server Faces Spezifikation zu erleichtern, gibt es die Basisklasse `javax.faces.component.UIComponentBase` für GUI Komponenten Implementierungsklassen. Diese ermöglicht es dem Komponentenentwickler, sich auf die wesentlichen implementierungstechnischen Details der neuen GUI Komponente zu konzentrieren, ohne sich um die folgenden Basiseigenschaften –und services einer Komponente befassen zu müssen:

- Das Speichern und Wiederherstellen von gespeicherten Objekten von `UIComponents`, also aller Arten HTML GUI Komponenten aus dem GUI Komponentenset. Objekte können entweder `java.util.Listen` sein, `javax.faces.component.StateHolder` Derivate, sprich wiederum `UIComponents` oder andere Objekte, die dann aber erzwungenermaßen einen Defaultkonstruktor haben müssen
- Das Aufrufen des per Tagattribut deklariterten Validators für die GUI Komponente
- Das Ermitteln des passendes Renderers und das Anstoßen der Dekodierung, sprich das Speichern von Benutzereingaben in einem Attribut einer von `UIInput` abgeleiteten HTML Komponentenimplementierungsklasse. Dies ist nur für GUI Komponenten relevant, die die Dateneingabe von Benutzern ermöglichen, wie z.B. Textfelder oder Textareas
- Die Initiierung des Sendens von Ereignissen an die jeweilige GUI Komponente

- Der Zugriff auf Factes und Kindelemente der GUI Komponente
- Die Verfügbarkeit und Speicherung von `ValueBinding` Ausdrücken unter Angabe eines Namensschlüssels

Der Rendering Framework um die `UIComponentBase` Klasse herum besteht aus folgenden Teilen:

- Den Implementierungsklassen für die Standard HTML GUI Komponenten in den Paketen `javax.faces.component` und `javax.faces.component.html`.
- Den abstrakten Basisklassen im Paket `javax.faces.render` für: `Renderer` Implementierungen, `RenderKit` Implementierungen, `RenderKitFactory` Implementierungen und einem `ResponseStateManager`.
  Ein `RenderKit` ist eine Art Fabrikklasse für `Renderer` Instanzen. Ein JSF Provider kann für verschiedene Markup Language Sprachen, linguale Sprachen oder Browsertypen ein passendes `RenderKit` implemenieren. Die `RenderKitFactory` verwaltet programmiertechnisch alle `RenderKit` Objekte einer JSF Implementierung
  Der `ResponseStateManager` ist eine Hilfsklasse für die Generierung des HTML Response Codes mittels des vom JSF Implementierungshersteller genutzten Renderingmechanismus.
  In seiner Methode `ResponseStateManager.writeState(FacesContext, SerializedView)` wird der Status des `UIViewRoot` Objekts und damit rekursiv der Staus aller auf der Seite befindlichen GUI Elemente clientseitig in ein HTML `<hidden>` Element gespeichert und steht somit für die Requestbarbeitung zu Verfügung.
- Den Implementierungen der abstrakten Klassen des `javax.faces.render` Pakets des jeweiligen Herstellers. Bei der JSF Referenzimplementierung von Sun liegen die Implementierungsklassen in `com.sun.faces.renderkit`.
- Allen `javax.faces.render.Renderer` Ableitungen im Paket `com.sun.faces.renderkit.html_basic`. Jeder `Renderer` der für eine spezifische HTML GUI Komponente benötigt wird, ist hier zu finden.

Jedes `javax.faces.render.Renderer` Derivat selbst, oder eine Basisklasse muss für die Encodierung einer HTML GUI Komponente, d.h. das Generieren des HTML Codes aus einer Java HTML – Tagimplementierungsklasse heraus drei Callbackmethoden implementieren:

```
Renderer.encodeBegin(FacesContext, UIComponent)
Renderer.encodeEnd(Faces Context, UIComponent)
Renderer.endcodeChildren(FacesContext, UIComponent)
```

Die erste der aufgeführten Methoden schreibt den Beginn Tag mit allen Attributen und Kindelementen. Für die Kindelemente bedient sie sich der `endcodeChildren()` Methode. Die `encodeEnd()` Methode schreibt den schließenden Tag des jeweiligen HTML Elements.

2.2.4 Event gesteuertes Aktion Handling – JSF Strategie des MVC Musters

Der gesamte JSF Event handling Mechanismus wird von der Spezifikation abgedeckt und ist somit weitgehend, abgesehen von der Implementierung der Listenerklassen, hersttellerunabhängig. Das bedeutet, dass sich ebenfalls weder Webapplikationsprogrammierer, noch GUI Komponentenentwickler um diese Internas kümmern müssen. Sie müssen gegebenenfalls nur die Konventionen einhalten, um den Event Mechanismus zu benutzen.

Alle Klassen, die das JSF Event handling bilden, befinden sich im Paket `javax.faces.event`. Jedes Event handling besteht vier Bausteinen:

1. Ereignisse, deren Vielfältigkeit durch eine Vererbungshierarchie abgebildet wird. Jedes Ereignis erbt mindestens von einer Eventbasisklasse. In JSF, wie auch im Swing GUI Framework ist dies die Klasse `java.util.EventObjekt`.
2. Event Listener, bei dem jeder genau einem Ereignistyp dediziert zugeordnet ist. Zwischen Ereignis und Listener besteht eine 1:1 Beziehung, wobei dem Listener ein ihm entsprechendes Eventobjekt in seiner Callbackereignisbearbeitungsmethode mitgegeben werden muss.
3. Objekte über die Ereignisse ausgelöst werden oder die selbst Ereignisse ausgelösen. Jedem Ergeignisobjekt muss das Quellereignisauslösungsobjekt übergeben werden.
4. Eine Event Queue, in der Ereignisse gespeichert, und zur Bearbeitung abgeholt werden

Mit diesen vier Bausteinen und der Art und Weise deren Zusammenwirkens, ist der JSF Event handling Mechanismus genau der gleiche der in Swing verwendet wird.

Nun stellt sich die Frage, wie das an Swing angelehnte Event handling in JSF implementiert ist. Diese Frage wird am Beispiel eines `ActionEvents` geklärt, wenn der Benutzer einen Submit button klickt.

Alles beginnt mit einer JSP deren `view/form` Element unter anderem am Schluss ein `<commandButton>` Element enthält. Auf einer entsprechenden JSP kann dies so aussehen:

```
<h:commandButton value="#{login}" action="#{loginBean.actionMethod}"/>
```

In dem generierten Servlet dieser JSP wird daraus folgender Quelltext:

```
private boolean _jspx_meth_h_commandButton_0(JspTag _jspx_th_h_panelGrid_1, PageContext
_jspx_page_context) throws Throwable {

    CommandButtonTag _jspx_th_h_commandButton_0 = TagHandlerPool.get(CommandButtonTag.class);

    _jspx_th_h_commandButton_0.setPageContext(_jspx_page_context);
    _jspx_th_h_commandButton_0.setParent((javax.servlet.jsp.tagext.Tag)_jspx_th_h_panelGrid_1);
    _jspx_th_h_commandButton_0.setValue("#{login}");
    _jspx_th_h_commandButton_0.setAction("#{loginBean.actionMethod}");

    int _jspx_eval_h_commandButton_0 = _jspx_th_h_commandButton_0.doStartTag();
    if (_jspx_th_h_commandButton_0.doEndTag() == javax.servlet.jsp.tagext.Tag.SKIP_PAGE)
      return true;

    TagHandlerPool.reuse(CommandButtonTag);
    return false;
}
```

Abbildung 6: Quelltext eines generierten Servelts für einen Command Button auf einer JSF.
Die folgenden Erläuterungen stellen den Zusammenhang zwischen Tagattributen auf der JSP und den korrespondierenden Methodenaufrufen der Tag Repräsentationsklassen dar.

Besonderes Augenmerk hinsichtlich des event processing ist auf den Aufruf der Methoden `CommandButtonTag.doStartTag()` und `CommandButtonTag.doEndTag()` zu richten (im Listing violett markiert). Die `doStartTag()` Methode ist Ausgangspunkt für den Aufruf des für dieses GUI Element passenden Renderers, der in seiner `ButtonRenderer.encodeBegin(FacesContext, UIComponentBase)` den HTML Code schreibt. Genau nach demselben Schema erfolgt diese Encodierung für jedes andere HTML GUI Element, das auf dieser JSP zu finden ist. Am Ende der vollständigen Encodierung der JSP ist der CommandButton bereit, auf einen Klick das Feuern eines `ActionEvents` anzustoßen. Für die genaue Aufrufhierarchie der Encodierung des `<commandButton>` Elements siehe Anhang 5 B.

Erfolgt nun ein Klick auf den CommandButton wird der Event processing Mechanismus für ein `ActionEvent` in Gang gesetzt. Dieser besteht aus zwei Teilen, wobei der erste Teil aus zwei

Unterteilen besteht, der Erzeugung des ActionEvent Objekts und dessen Speicherung in eine Event Queue für die spätere Ereignisbearbeitung. Die ActionEvent Objekterzeugung wird durch die zweite Phase des Request Processing Lifecycle, in der ApplyRequestValuesPhase.execute(FacesContext) Methode initiiert.

Initiierung bedeutet die UIViewRoot.processDecodes(FacesContext) Methode aufzurufen. Das Decoding kann als Gegenrichtung zum Element Encoding verstanden werden, da im Gegensatz zum Encoding nicht aus Tag Library Elementen HTML Code generiert wird, sondern die Informationen aus den HTML Quelltext Tagattributen in einem Javaobjekt verpackt werden und daraus dann weitere Objekte, in diesem Fall die Ereignisobjekte, erzeugt werden. Wie auch beim Encoding wird für jedes HTML GUI Element Repräsentationsobjekt wie z.B. javax.faces.component.UIInput oder javax.faces.component.html.HtmlCommandButton der korrespondierende Renderer ermittelt. Für den hier diskutierten CommandButton wird für das Decoding aufgerufen ButtonRenderer.decode(FacesContext) und dort schließlich wird das entscheidende ActionEvent Objekt erzeugt. Damit ist der erste Unterteil des ersten Teils des Event processing abgeschlossen.

Direkt im Anschluss wird das ActionEvent in einer Event Queue gespeichert. Diese befindet sich in Form eines java.util.List[] Objekts im UIViewRoot Objekt. Es wird nicht sofort oder direkt an einen ActionListener weitergeleitet und verarbeitet. Dies passiert erst im zweiten Teil Event processing. Der Weg wie letztlich jedes Ereignisobjekt egal welchen Typs bei der UIViewRoot.queueEvent(FacesEvent) Methode landet, läuft über die bekannte Rekursion. Jedes UIComponent Objekt erbt mindestens von UIComponentBase. Die dortige queueEvent(FacesEvent) Methode ist so implementiert, dass sie immer die UIComponent.getParent().queueEvent(FacesEvent) Methode des Elternelements der gegenwärtigen GUI Komponente aufruft, solange bis UIComponent dann das UIViewRoot Objekt ist. Dort wird dann das FacesEvent abhängig von seiner Lifecyle Phasenzugehörigkeit, oder wenn es sich um ein benutzergetriggertes Ereignis handelt, in einer anderen Liste, der EventQueue List[] events abgelegt.

Der Eventverteil- und Bearbeitungsschritt folgt in der fünften und vorletzten Phase des Request Processing Lifecycle. In ihrer execute(FacesContext) Methode ruft die Invoke ApplicationPhase UIViewRoot.processApplication(FacesContext) auf. In dieser Methode werden alle Ereginisse, die in allen Phasen davor erzeugt und gespeichert wurden an alle ihre Empfänger - UIComponents gesendet. Angetrieben wird dieser Schritt durch UIViewRoot.broadcastEvents(FacesContext, PhaseId). Es werden zwei Arten von Ereignissen unterschieden: Lifecycle und Benutzer ausgelöste events. Gekennzeichnet werden die Ereignisse die durch eine Benutzerinteraktion ausgelöst wurden mit der PhaseId PhaseId.ANY_PHASE. Zuerst werden diese abgearbeitet.

Mit dem Aufruf UIComponent.broadcast(FacesEvent) beginnt der zweite Schritt der Ereignisbehandlung. Aus der EventQueue werden alle Ereignisse des Kennzeichners PhaseId.ANY_PHASE abgeholt: List = EventQueue[PhaseId.**ANY_PHASE**.getOrdinal()]. Für jedes Ereignis wird dessen Quelle abgefragt: UIComponent = FacesEvent.getComponent() und die resultierende Auslöser GUI Komponente leitet dann mit UIComponent.broadcast(FacesEvent) die Bearbeitung des Ereignisobjekts ein.

Die Bearbeitung des Ereignisses erfolgt in zwei wesentlichen Schritten: Erstens, das Ermitteln des richtigen Event Listeners. Ist das Ereignis ein ActionEvent, wird mit ActionListener.processAction(ActionEvent) der zweite Schritt eingeleitet, das Aufrufen der Ereignisbearbeitungsmethode der richtigen Beanklasse für die View JSP. Dies läuft folgendermaßen ab: Die Ereignisquelle muss ein UIComponent sein, das das Interface ActionSource implementiert. Dies ist deswegen klar, weil nur UIComponents diesen Typs ActionEvents auslösen können.

Das `ActionSource` Interface bietet eine Methode `ActionSource.getAction()`, die ein `MethodBinding` zurückliefert, das auf eine Methode eines Beanobjektes referenziert. Dies kann z.B. so aussehen:

```
<h:commandButton value="OK" action="#{bean.actionMethod}">
```

Das `MethodBinding` der `ActionSource` beinhaltet den Attributwert des Attributes `action`. Wird dieses `MethodBinding` mit `MethodBinding.invoke(FacesContext, Object[] params)` ausgeführt, liefert es ein outcome String zurück. Dieses outcome ist ein Schlüssel für die nächste View Seite auf die nach Abarbeitung der Ereignisbehandlungsmethode des Beanobjektes verzweigt werden soll. Es steht in einer `faces-config.xml` unter folgendem Element:

```
<navigation-rule>
  <from-view-id>/view/source/sourcePage.jsp</from-view-id>
  <navigation-case>
    <from-outcome>success</from-outcome>
    <to-view-id>/view/destination/destinationPage.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

Ist der Wert des outcome Strings `success`, ist klar, dass die `NavigationHandler` Implementierung von der aktuellen `/view/source/sourcePage.jsp` auf die Seite `/view/destination/destinationPage.jsp` springen muss. Diese Aufgabe delegiert der `ActionListener` mit dem Aufruf `NavigationHandler.handleNavigation(FacesContext, String outcome)`.

Nachdem alle Benutzerereignisse erledigt sind, kommen die Lifecycle Events an die Reihe. Der Aufruf zur Delegation der Ereignisbearbeitung an das Ergeinisquell `UIComponent` Objekt in der `UIViewRoot.broadcastEvents(FacesContext, PhaseId)` Methode ist zwischen Lifecycle Events und denen vom Benutzer ausgelösten Ereignissen bis auf eine Kleinigkeit identisch. Aus der EventQueue wird nicht die Ereignisliste `List = EventQueue[PhaseId.`**`ANY_PHASE`**`.getOrdinal()]` abgeholt, sondern die der jeweiligen Request Processing Lifecycle Phase mit `List = EventQueue[`**`PhaseId`**`.getOrdinal()]`. `PhaseId` wurde als Parameter von `UIViewRoot.broadcastEvents(FacesContext, PhaseId)` übergeben. Mit der Abarbeitung der Request Processing Lifecycle Eventobjekte ist das Event Processing der Java Server Faces am Beispiel eines `ActionEvents` abgeschlossen. Bei jedem erneuten Klick auf einen `<h:commandButton>` oder `<h:commandLink>` Element wird dieser Teilschritt des Request Processing wieder von vorne durchlaufen.

2.3 Die Initialisierung einer JSF Webapplikation

Der Startpunkt einer JSF Webapplikation ist wie bei jeder Servlet API basierten Anwendung die init Methode eines Servlets. Bei den JSF beginnt es mit `FacesServlet.init(ServletConfig)`. Dort sind drei wesentliche applikationsweite Objekte zu instanziieren: das `javax.faces.application.Application`, die `javax.faces.context.FacesContextFactory` und das `javax.faces.lifecycle.Lifecycle` Objekt.

Für die Erzeugung dieser zentralen und komplexen Objekte gibt es im JSF Framework ein aus zwei Teilen bestehendes Konstrukt: dem `javax.faces.FactoryFinder` und den Fabrikklassen `ApplicationFactory`, `FacesContextFactory`, `LifecycleFactory` und `RenderKitFactory`. Es wird also deutlich, dass für jedes globale applikationsweit gültige Objekt eine Fabrikklasse existiert, die die Erzeugung und Verwaltung ihrer hergestellten Objekte übernimmt. Es ist dem Applikationsentwickler nicht möglich, diese Objekte selbst zu erzeugen. Falls er aber Dienste dieser Objekte in Anspruch nehmen möchte, kann er über den `FacesContext` eine Referenz auf

das gewünschte globale Objekt erhalten. Den `FacesContext` selbst stellt die Methode `FacesContext.getInstance()` bereit und somit ist dieser von überall aus erreichbar.

Der `FactoryFinder` ist seiner Funktion eine Metafabrik, also eine Fabrik die Fabriken verwaltet und verfügbar macht. Wann immer es um die Kapselung der Erzeugung und architekturtechnische Verwaltung zentraler Datenobjekte geht, ist das Fabrikmuster eine Standardstrategie, die in Frameworks und komplexen Anwendungen verwendet wird.

Um von (Meta) Fabriken die gewünschten Objekte erzeugt zu bekommen, gibt es generell drei Wege der Fabrik mitzuteilen, welches Produkt benötigt wird. Die einfache Parameterübergabe anhand eines Identfikationsstrings –oder Integers, bzw. eines Strings, der den vollklassifizierenden Namen der gewünschten Resource enthält, ist die erste Variante. Die ähnliche, etwas anspruchsvollere Alternative ist ein dynamischer Reflectionmechanismus. Die dritte Möglichkeit ist die Verwendung einer Registry, wie sie im Betriebssystem Windows verwendet wird. Auch die `struts-config.xml` oder die `faces-config.xml` kann als Registry bezeichnet werden, da das `ActionServlet` diese zum Erzeugen von den dort eingetragenen `Action` und `ActionForm` Instanzen verwendet, bzw. JSF die `faces-config.xml` zur Instanziierung von Modelklassen nutzt.

Die JSF Referenzimplementierung verwendet für die globalen Fabrikobjekte die erste Variante, was aufgrund der Einfachheit angebracht ist, da in der vorliegenden Version nur die vier erwähnten Fabrikobjekte existieren, die jeweils nur ein Objekttyp verwalten für die Modelklassen die dritte Variante. Um das `Application`, `Lifecycle` und das `FacesContext` Objekt zu erhalten, übergibt das `FacesServlet` dem `FactoryFinder` dessen die Fabrik indentifizierenden Identifikationsstring. Für das `Application` Objekt sieht dies so aus:

```
ApplicationFactory = FactoryFinder.getFactory(FactoryFinder.APPLICATION_FACTORY).
```

Im zweiten Schritt wird die erhaltene Fabrikklasse beauftragt das Zielobjekt herzustellen:

```
Application = ApplicationFactory.getApplication().
```

Entsprechend das gleiche passiert mit dem `Lifecycle` und `FacesContextFactory` Objekt. Am Beispiel des `Application` Objekts wird der Verlauf dessen Instanziierung über die `Application Factory` Implementierung erläutert:

Die für die Erzeugung der `ApplicationFactory` zuständige Methode `FactoryFinder.get Factory(factoryClassName)` besorgt sich zunächt einen `ClassLoader` mittels `ClassLoader = Thread.currentThread().getContextClassLoader()`. Der gewonnene `Classloader` und der `factoryClassName` Parameter werden an die Methode `FactoryFinder.get ImplementationInstance(ClassLoader, factoryKey, List)` weitergeleitet.. Der `factoryKey` Parameter enthält einen Wert der einem der Fabrikklasseninstanziierungsschlüsseln entspricht. Ein Instanziierungsschlüssel wird in einer JSF Implementierung für jede Fabrikklasse für deren Identifizierung festgelegt. Für die `ApplicationFactory` ist dies der Wert `javax. faces.application.ApplicationFactory` für die Konstante `public static final FactoryFinder.APPLICATION_FACTORY`. Der `List` Parameter enthält möglicherweise eine Liste von Implementierungsklassennamen für Fabrikklassen.

Diese Methode geht nun folgendermaßen vor, um ein passendes Fabrikklassenobjekt zu erzeugen. Es wird zuerst untersucht, ob im `List` Parameter ein oder mehrere vollklassifizierende Klassennamen enthalten sind. Ist dies der Fall wird der letzte Listeneintrag zur Fabrikklassenerzeugung genutzt. Dieser letzte Listeneintrag wird `FactoryFinder.getImplGivenPrevious Impl(ClassLoader, factoryKey, implName, Object)` übergeben. Nun gibt es drei Fälle, wie die Fabrikklassen instanziiert werden können.

Im einfachsten Fall bei dem nur der übergebene factoryClassName Parameter zur Verfügung steht, ruft FactoryFinder.getImplementationInstance() die eigentliche Reflectionerzeugungsmethode FactoryFinder.getImplGivenPreviousImpl(ClassLoader factory Key, factoryImplClassName, Object factoryImpl) auf, die dann letztendlich das Fabrikklassenimplementierungsobjekt zurückliefert.

Der zweite Fall besteht darin, eine Datei /META-INF/services/<Fabrikklassenname> im Webapplikationshauptverzeichnis als .properties Datei zu behandeln, falls der Applikationsprogrammierer eine solche erstellt hat. Diese Datei muss einen Eintrag mit einem vollklassifizierenden Klassennamen für die eigens implementierte Fabrikklasse enthalten.

Beim dritten Fall wird davon ausgegangen, dass die Liste der vollklassfizierenden Fabrikimplementierungsklassennamen aus FactoryFinder.getImplementationInstance(Class Loader, factoryKey, List) länger als eins ist. Hier wird einfach das letzte Listenelement zur Instanziierung der Fabrikklasse verwendet und wieder an die eigentliche Objektinstanziierungsmethode weitergeleitet.

Abschließend zum Fabrikklassenerzeugungsvorgang ist noch die Verfahrensweise der Methode FactoryFinder.getImplGivenPreviousImpl(ClassLoader, factoryKey, factoryImplClassName, Object factoryImpl) zu klären. Wichtig ist der letzte Parameter. Ist er nicht null, versucht die Methode einen ein - Argumentkonstruktor für die zu ladende Klasse für den vollklassifizierenden Klassennamen im Parameter factoryImplClassName zu finden:

```
Class = ClassLoader.loadClass(factoryImplClassName)
Class[0] = getFactoryClass(ClassLoader, factoryKey)
Constructor = Class.getConstructor(Class[0])
Object = Constructor.newInstance(factoryImpl)
return Object
```

Ist der Parameter factoryImpl null, wird einfach auf den Defaultkonstruktor der Fabrikimplementierungsklasse zurückgegriffen und so die Fabrikklasse erstellt:

```
Class = ClassLoader.loadClass(factoryImplClassName)
Object = Class.newInstance()
return Object
```

Da nun die Implementierung der ApplicationFactory zur Verfügung steht, kann das eigentliche Ziel, das Application Objekt mit ApplicationFactory.getApplication() erzeugt und gesichert werden. Im Konstruktor der Application Objektimplementation passieren zwei Dinge: Die Erzeugung des Application Helferobjekts com.sun.faces.application.ApplicationAssociate und die Initialisierung mehrerer HashMaps. In diesen HashMaps werden folgenden Informationen hinterlegt:

- ValueBinding Objekte
- Vollklassifizierende Klassennamen von UIComponent Objekten
- Vollklassifizierende Klassennamen von javax.faces.convert.Converter Derivaten
- Vollklassifizierende Klassennamen von javax.faces.validator.Validator Derivaten

Innerhalb von ApplicationAssociate werden zwei HashMaps für die Sicherung der Inhalte der faces-config.xml mit ihren zwei Hauptelementen managed-bean und navigation-rule angelegt. Die zwei HashMaps beinhalten je die Repräsentationsobjekte dieser xml Elemente com.sun.faces.config.beans.ManagedBeanBean sowie dessen Fabrik- bzw. Wrapperobjekt com.sun.faces.config.ManagedBeanFactory und com.sun.faces.application.Config NavigationCase.

Schritt 1: `Initilisierungsvorbereitung`

Ähnlich wie beim Struts Framework die `struts-config.xml` in ein Javaobjekt überführt werden musste, sind für die `faces-config.xml` Inhalte und Elemente ebenfalls Javarepräsentationsobjekte zu erzeugen, um sie danach in Datenstrukturen wie Listen, Maps oder Tables aus dem Java Collection API speichern zu können. Diese Datenstrukturen müssen in globalen Frameworkobjekten verfügbar sein, damit Objekte, die relevante Informationen beinhalten, bei Bedarf seitens des Benutzers und des Frameworks selbst, gekapselt von der internen Speicher- und Verwaltungsapplikationslogik zur Verfügung stehen.

Genau diese Anforderung erfüllen die globalen JSF Frameworkobjekte `javax.faces.context.FacesContext`, `javax.faces.context.ExternalContext` und `javax.faces.application.Application`. Bevor die für die jeweilige Webanwendung wichtigen Daten in den Objektspeicherdatenstrukturen der eben genannten globalen Objekten bereit stehen, muss es vor dem Aufruf der `FacesServlet.init(ServletConfig)` Methode einen Mechanismus geben, der alle wichtigen resource xml Dateien parst, xml Elementrepräsentationsobjekte erzeugt und deren Inhalte in den Datenstrukturen der globalen JSF Frameworkobjekte ablegt.

Diese Aufgaben übernimmt die Klasse `com.sun.faces.config.ConfigureListener`, die das Interface `javax.servlet.ServletContextListener` implementiert. Der Webserver instanziiert diese Klasse unter Verwendung folgenden Eintrags im `web.xml` Deploymentdeskriptor der Webanwendung.

```
<listener>
  <listener-class>com.sun.faces.config.ConfigureListener</listener-class>
</listener>
```

Die Funktionalität des `ConfigureListener` ist in der Methode `ConfigureListener.contextInitialized(ServletContextEvent)` und `ConfigureListener.contextDestroyed(ServletContextEvent)` aus dem `ServletContextListener` Interfaces und einigen Hilfsmethoden hinterlegt.

Das Hauptziel des `ConfigureListener` ist es, die Daten aus der `faces-config.xml` in in das `com.sun.faces.config.beans.FacesConfigBean` Objekt zu überführen. Das `FacesConfigBean` Objekt ist das Pendant zum `ModuleConfig` Objekt des Struts Framework. In dieser Eigenschaft besitzt es erstens `java.util.TreeMap` Objekte, die die folgende Baumstrukturen von xml Elementen speichern: `<managed-beans>`, `<navigation-rule>`, `<referenced-bean>`, `<render-kit>`, `<validator>`, `<converter>`. Zweitens werden als einfaches Klassenattribut Konfigurationsbeanobjekte für die globalen Frameworkobjekte `Lifecycle` und `Application` gespeichert, sowie ein `FactoryBean` Objekt, das Factoryobjekte von JSF, also `RenderKitFactory`, `FacesContext Factory`, `ApplicationFactory` und `LifecycleFactory` Instanzen beinhaltet.

Wie die Initialisierung aller Beanrepräsentationsobjekte der `faces-config.xml` Elemente von statten geht, und wie und wo diese dem JSF Framework zugänglich gemacht werden, ist Gegenstand der folgenden Betrachtungen.

Alles beginnt damit, dass der Servlet Container, bzw. der Webserver beim Hochfahren alle Webapplikationen, die er in einem `webapps` Verzeichnis oder als `<Context>` Elementeintrag in seiner `server.xml` findet. Befindet sich unter diesen Webapplikationen eine JSF Webanwendung, untersucht er dessen Deployment Deskriptor bei einem `listener/listener-class` Eintrag auf eine `javax.servlet.ServletContextListener` Implementierungsklasse. Benutzt der Webentwickler die JSF Referenzimplementierung von Sun muss der Wert dieses Elements `com.sun.faces.config.ConfigureListener` sein. Wurde der Tomcat Webserver fündig, ruft

er die ConfigureListener.contextInitialized(ServletContextEvent) Methode auf und startet somit den Konfigurationsprozess. Mit dem erhaltenen ServletContextEvent hat der ConfigureListener direkten Zugriff auf den ServletContext der als Schnittstelle zwischen Webserver und Webentwicklungsframework genauso wie bei Struts auch in JSF eine entscheidenden Rolle als Standardinformationsspeicherobjekt spielt.

Schritt 2: Initialisierung eines Digesters zum Parsen von xml Konfigurationsresourcen

Der erste Schritt zum Erstellen eines komplett konfigurierten FacesConfigBean Objekts ist die Erstellung eines Digesters aus dem Paket org.apache.commons.digester. Diese erfolgt in einer Hilfsmethode ConfigureListener.digester(boolean validate). Genauso wie bei der Digesterkonfiguration beim Struts Framework wird dem Digester ebenfalls in dieser Methode ein org.apache.commons.digester.RuleSet Objekt mitgegeben, das genau den Anforderungen entspricht, um die faces-config.xml zu parsen. Dazu gibt es das Faces ConfigRuleSet, das dem Digester alle org.apache.commons.digester.Rule Objekte aus dem Paket com.sun.faces.config.rules hinzufügt. Diese Rule Derivate bilden je die xml Hauptelemente unter dem Wurzelelement <faces-config> der faces-config.xml ab. Realisiert wird dies mit der Methode ConfigRuleSet.addRuleInstances(Digester). Innerhalb dieser Methode werden dem Digester alle faces-config.xml Elemente entsprechend der web-facesconfig_1_1.dtd mittels zweier Methoden Digester.addRule(String element,Rule) und Digester.addCallMethod(String element, String method, int) hinzugefügt. Die letzte der beiden genannten Methoden ist lediglich eine Hilfsmethode mittels derer eine CallMethodRule erzeugt wird und diese mit der ersten Methode anschließend wie jede andere Rule in den Cache von Rule Objekten, der org.apache.commons.digester. RulesBase gesichert wird.

Für viele der faces-config.xml Elemente wird dem Digester eine extra von org. apache.commons.digester.Rule abgeleitete Implementierungsklasse mitgeliefert. Diese übernimmt beim Aufruf in ihren Callbackmethoden dann die eigentliche Arbeit, die faces-config.xml Elementrepräsentationsobjekte zu erzeugen und in das FacesConfigBean Objekt zu hinterlegen. Beispiele für diese JSF Elementrepräsentationsobjekte sind NavigationRule Rule und NavigationCaseRule aus dem Paket com.sun.faces.config.rules.

Anhand des <navigation-rule> Elements und deren Kindelemente wird demonstriert, wie anhand des ConfigRuleSet der Digester das FacesConfigBean mit Daten füllt. Dieses Verfahren ist vom Prinzip her identisch wie für die struts-config.xml Elemente beim Struts Framework und damit natürlich auch für alle anderen Elemente der faces-config.xml.

In der Methode FacesConfigRuleSet.addRuleInstances(Digester) beginnt die Aufarbeitung der Inhalte eines <navigation-rule> Elements mit dem Aufruf Digester. addRule("faces-config/navigation-rule", new NavigationRuleRule()). Laut der web-facesconfig_1_1.dtd besitzt das <navigation-rule> Element die zwei Unterelemente <from-view-id> und <navigation-case>. Daher müssen die Regeln für diese Elemente dem Digester ebenfalls bekannt gemacht werden. Da es sich bei bei dem <from-view-id> um kein weiter geschachteltes Element handelt und da es nur dazu dient, als Kindelement von <navigation-rule> ein Attribut von dessen NavigationRuleRule zu setzen, ist das <from-view-id> Element mit der CallMethodRule dem Digester hinzuzufügen. Aus Digester. addCallMethod("faces-config/navigation-rule/from-view-id", "setFromViewId", 0) im FacesConfigRuleSet wird im Digester, Digester.addRule(String xmlElement, new CallMethodRule(String methodName)).

Das Element <navigation-case> besitzt Kindelemente und damit eine komplexere Struktur und muss, da es speziell behandelt werden muss, eine eigene Processing Rule, die NavigationCaseRule besitzen. Die Kindelemente sind allesamt Blattelemente und da auch sie lediglich die Funktion des Setzens von Attributen ihres Kindelements haben, können sie wie das <from-view-id> Element mit der CallMethodRule erledigt werden. Die Stelle im Quelltext von FacesConfigRuleSet sieht analog zum <navigation-rule> Element aus:

```
Digester.addRule("faces-config/navigation-rule/navigation-case",
                 new NavigationCaseRule());
Digester.addCallMethod("faces-config/navigation-rule/navigation-case/from-
                       action",setFromAction",0) ...
```

Nun gilt es noch zu klären, wie die Processing Regeln NavigationRuleRule und NavigationCaseRule funktionieren, um die Inhalte der entsprechenden faces-config.xml Elemente im FacesConfigBean zu speichern. In beiden Rules wie auch bei jeder andernen Rule befindet sich die Implementierungslogik in den zwei Methoden Rule.begin(nameSpace, elementName, Attributes) und Rule.end(nameSpace, elementName).

In NavigationRuleRule.begin(nameSpace,elementName, Attributes) wird zunächst das oberste Objekt vom Stack des Digesters geholt, das FacesConfigBean. Es wurde nach seiner Instanziierung in ConfigureListener.contextInitialized(ServletContextEvent) in der ConfigureListener.parse(Digester, URL, FacesConfigBean) Methode auf den Stack des Digesters mittels Digester.push(Object) abgelegt. Nachdem der Digester seinerseits das Parsen mit Digester.parse(InputSource) gestartet hat, steht das FacesConfigBean nun für jedes Rule Objekt zur Verfügung. Da der Digester ein eventgesteuerter SAX Parser ist, der im Gegensatz zu einem DOM Parser keine Java xml Element Objekte zwischenspeichert ruft der Eventmechanismus des SAX APIs die Callbackmethoden Rule.begin(nameSpace, element Name, Attributes) und Rule.end(nameSpace, elementName) aller Rule Objekte auf, die ihm in FacesConfigRuleSet.addRuleInstances(Digester) gegeben wurden.

Nachdem in NavigationRuleRule.begin(nameSpace, elementName, Attributes) das FacesConfigBean vom Digester Stack mit Digester.peek() geholt, aber nicht gelöscht wurde, wird ein für das xml Element <navigation-rule> zuständige Javaobjekt NavigationRuleBean per Reclection erzeugt und abermals auf dem Stack des Digesters abgelegt.

Der nächte Bearbeitungsschritt erfolgt in der NavigationRuleRule.end(nameSpace, elementName) Methode. Da nun das eben erzeugte NavigationRuleBean auf dem Digester Stack ganz oben liegt, kann es mit Digester.pop() vom Stack geholt und gelöscht werden und steht damit für die weitere Bearbeitung zur Verfügung. Da nun wieder das FacesConfigBean wieder an oberster Stelle des Digester Stacks liegt, steht auch dieses wieder mit Digester.peek() bereit.

Im nächsten Schritt folgt eine für die NavigationRuleRule individuelle Logik, nachdem zuvor Stack Schiebe- und Löschoperationen ausgeführt wurden, die in allen Rule Derivaten vorkommen. Es kann vorkommen, dass der Entwickler fälschlicherweise zwei <navigation-rule> Elemente in der faces-config.xml deklariert, die beide denselben Wert beim Unterelement <from-view-id> haben. Ist dies der Fall, müssen diese beiden <navigation-rule> Elemente gemerged werden, d.h. sie müssen in ein einziges Element überführt werden. Genau dies leistest der Aufruf FacesConfigBean.getNavigationRule(String viewId).

Der viewId Parameter stammt von dem NavigationRuleBean Objekt, das in der NavigationRuleRule.begin(nameSpace, elementName, Attributes) Callbackmethode gerade erzeugt und auf dem Stack abgelegt wurde. Gibt FacesConfigBean.getNavigation

Rule(String viewId) null zurück ist sicher, dass alles in Ordnung ist und das Navigation RuleBean Objekt dem FacesConfigBean mit FacesConfigBean.addNavigationRule(NavigationRuleBean) hinzugefügt werden kann.

Ist der angesprochene Fehler passiert besteht das Mergen darin, einfach die NavigationCase Bean Objekte des neuen NavigationRuleBean Objekts dem alten hinzuzufügen, das ja aufgrund des identischen <from-view-id> Elements dem vorherigen NavigationRuleBean gleicht.

Das Ergebnis: das neue NavigationRuleBean wurde dem FacesConfigBean hinzugefügt und liegt nun ganz oben auf dem Stack des Digesters und kann nun mit der NavigationCase Rule mit NavigationCase Objekten gefüllt werden. Wie das funktioniert wird in den nächsten Absätzen behandelt.

Die einzige Funktionalität der NavigationCaseRule besteht darin, ein Navigation CaseBean einem NavigationRuleBean hinzuzufügen. Dazu wird in der NavigationCase Rule.begin(name Space, elementName, Attributes) Methode zuerst per Reflection das Objekt com.sun.faces.config.beans.NavigationCaseBean erzeugt und auf dem Stack des Digesters abgelegt:

```
Class = Digester.getClassLoader().loadClass(com.sun.faces.config.beans.NavigationCaseBean)
NavigationCaseBean ncb = (NavigationCaseBean) Class.newInstance()
Digester.push(NavigationCaseBean)
```

In NavigationCaseRule.end(nameSpace,elementName) wird das soeben erzeugte und auf Digester Stack abgelegte NavigationCaseBean geholt und gelöscht: NavigationCase Bean = (Navigation CaseBean)Digester.pop(). Zusätzlich wird das zuvor geparste NavigationRuleBean mit NavigationRuleBean = Digester.peek() abgeholt. Im letzten Schritt kann nun ausgeführt NavigationRuleBean.addNavigationCase(NavigationCase Bean) werden, womit dem NavigationRuleBean eine weiterer Navigation Case hinzugefügt wurde.

Was einem NavigationRuleBean Objekt noch fehlt, sind Werte für seine Attribute **fromAction**, fromOutcome, toViewId und redirect. Dies wird im FacesConfigRuleSet eingeleitet mit Digester.addCallMethod("faces-config/navigation-rule/navigation-case/**from-action**", "**setFromAction**", 0)
Gleiches passiert analog mit den anderen Attributen.

Ergebnis: Es ist nun offengelegt, wie alle Daten der für den JSF Webanwendungsentwickler wichtigsten faces-config.xml Elemente <navigation-rule> und <navigation-case> in das FacesConfigBean gelangen. Es genügt bereits die Anwendung dieses Wissens, um das Navigationsverhalten der Webapplikation zu steuern, da das JSF Framework anhand dieser Informationen den Teil der Seitenverzweigung des Request Processing vollständig abnimmt.

Nachdem durch die Methode ConfigRuleSet.addRuleInstances(Digester) dem Digester alle Rule Objekte übergeben wurden, ist dieser nun in der Lage die faces-config.xml zu parsen, womit dessen Konfiguration abgeschlossen ist. All die unter „Schritt 2" erläuterten Funktionalitäten erfolgen durch den Aufruf Digester.addRuleSet(RuleSet) in der Methode ConfigureListener.contextInitialized(ServletContextEvent). Neben den Inhalten der Resource faces-config.xml werden auch noch die Daten anderer Resource Dateien in das FacesConfigBean aufgenommen. Die Schritte die hierfür nötig sind, folgen jetzt.

Schritt 3: **Das Parsen der jsf-ri-config.xml**

Der zweite Schritt, der nach der Digester Konfiguration folgt, ist das Parsen der Datei com/sun/faces/jsf-ri-config.xml. Diese Datei enthält die vollklassifizierenden Klassennamen der Implementierungsklassen für die Referenzimplementierung. Dazu gehören:

- Alle Converter Klassen aus dem Paket javax.faces.convert
- Alle Objekte, die die Implementierung von javax.faces.application.Application besitzt, sprich ActionListenerImpl, NavigationHandlerImpl, PropertyResolverImpl, StateManagerImpl, VariableResolverImpl und ViewHandlerImpl
- Alle Factory Implementierungsklassen: ApplicationFactoryImpl, FacesContextFactory LifecycleFactoryImpl und RenderKitFactoryImpl
- Die Standard Validatoren der Referenzimplementierung aus dem Paket javax.faces.validator: DoubleRangeValidator, LengthValidator und LongRangeValidator

Wenn in der faces-config.xml ausschließlich <navigation-rule> und <managed-bean> Elemente vorkommen, sprich keine Angaben zu eigenen Implementierungsklassen, die die der Referenzimplementierung ersetzen, werden die eben genannten Defaultimplementierungen verwendet und in das FacesConfigBean übernommen.

Dazu wird in FacesConfigRuleSet.addRuleInstances(Digester) dem Digester die com.sun.faces.config.rules.ApplicationRule hinzugefügt:

```
Digester.addRule("faces-config/application", new ApplicationRule())
```

Die ApplicationRule bewirkt beim Parsen des Elements faces-config/application der Datei jsf-ri-config.xml, die Erzeugung eines com.sun.faces.config.beans.ApplicationBean Objekts und dessen Ablage auf dem Stack des Digesters mit Digester.push(ApplicationBean).

Anschießend folgen in FacesConfigRuleSet.addRuleInstances(Digester) die Anweisungen:

```
Digester.addCallMethod("faces-config/application/action-listener",
                        "addActionListener", 0);
Digester.addCallMethod("faces-config/application/navigation-handler",
                        "addNavigationHandler", 0);
Digester.addCallMethod("faces-config/application/property-resolver",
                        "addPropertyResolver", 0);
Digester.addCallMethod("faces-config/application/state-manager",
                        "addStateManager", 0);
Digester.addCallMethod("faces-config/application/variable-resolver",
                        "addVariableResolver", 0);
Digester.addCallMethod("faces-config/application/view-handler",
                        "addViewHandler", 0);
Digester.addCallMethod("faces-config/application/default-render-kit-id",
                        "setDefaultRenderKitId", 0);
```

Diese bewirken das Hinzufügen von org.apache.commons.digester.CallMethodRule Objekten in den Rule Speicher des Digesters. Wenn der Digester beim Parsen auf die Elemente trifft, die im ersten String Parameter der Methode Digester.addCallMethod(**String xmlElement**, String, int) hinterlegt sind, wird in der CallMethodRule das oberste Objekt auf dem Stack des Digesters geholt, sprich Object = Digester.peek(). Das Object ist in diesem Fall das angesprochene ApplicationBean.

Die CallMethodRule ruft per Reclection jeweils diejenige Methode des ApplicationBean Objekts auf, die im zweiten Parameter Digester.addCallMethod(String, **String method,**

int) steht. Bezogen auf diese Funktionalität sieht die Implementierung, die die CallMethod Rule mit der Hilfsklasse org.apache.commons.beanutils.MethodUtils realisiert, so aus, um dem ApplicationBean das com.sun.faces.application.ActionListenerImpl Objekt hinzuzufügen. In der Methode CallMethodRule.end() steht folgendes:

```
Object top = Digester.peek()  // Object instanceof ApplicationBean
MethodUtils.invokeMethod(ApplicationBean, "addActionListener", paramValues,
paramTypes)
```

Die Parameter paramValues und paramTypes sind in diesem Fall leere Arrays, da ursprünglich bei Digester.addCallMethod(String, String, int) der int Parameter den Wert 0 hatte, d.h. die beiden Arrays haben die Länge 0. In dieser MethodUtils Methode wird auf das übergebene ApplicationBean Objekt über Method = ApplicationBean.getClass().get Method("addActionListener", Class[] args) per Reflection die addActionListener Methode aufgerufen: Method.invoke(ApplicationBean, new Object[] { String arg1, ... }). Der String arg1 Parameter stimmt mit dem Wert com.sun.faces.application. ActionListenerImpl des Elements application/action-listenener der Datei jsf-ri-config.xml überein.

Ergebnis: Nach der Erzeugung des ApplicationBean Objekts und dessen Ablage auf dem Digester Stack, wurde dem ApplicationBean mittels ApplicationBean.addAction Listener(String className) der vollklassifizierende Klassenname des ActionListener Impl hinzugefügt. Nach dieser Aktion sprich beim Erreichen des Ende Tags des faces-config/application Elements der jsf-ri-config.xml, wird das ApplicationBean Objekt in ApplicationRule.end(String, String) wieder vom Stack des Digesters gelöscht: Object top = Digester.pop(), womit wieder das FacesConfigBean auf Position eins steht.

Genau nach diesem Strickmuster werden die weiteren Anweisungen in FacesConfigRule Set.addRuleInstances(Digester) nach Digester.addCallMethod("faces-config/ application/action-listener", "addActionListener", 0) abgearbeitet. D.h. es werden dem ApplicationBean noch das NavigationHandlerImpl, das PropertyResolverImpl, das StateManagerImpl, das VariableResolverImpl und das ViewHandlerImpl Objekt hinzugefügt. Jeweils mit der entsprechenden Methode ApplicationBean.addXYZ(String class Name). Damit ist der Arbeitsschritt

```
URL = Util.getCurrentLoader(ConfigureListener).getResource(RIConstants.JSF_RI_CONFIG)
ConfigureListener.parse(Digester, URL, FacesConfigBean)
```

abgeschlossen. Nun folgen diese gleichen Anweisungen für die Datei com/sun/faces/ standard-html-renderkit.xml.

Schritt 4: Das Parsen der standard-html-renderkit.xml

Die standard-html-renderkit.xml enthält die vollklassifizierenden Klassennamen aller Renderer Klassen für alle UIComponent HTML GUI Elemente der JSF Referenzimplementierung.

Wichtig zu beachten ist, dass diese Datei zweimal, also zweimal exakt mit demselben Dateinamen existiert. In der ersten Variante befinden sich ausschließlich DTD Entity Deklarationen. Diese Entitäten legen mit dem Namen unter dem sie aufgerufen werden den vollklassifizierenden Klassennamen einer Renderers für das korrespondierende HTML GUI Repräsentationsobjekt fest. So ist z.B. für den Standard HTML Submit Button die JSF Repräsentationsklasse javax.faces.component.html.HtmlCommandButton, die Rendererklasse com.sun.faces. application.renderkit.html_basic.ButtonRenderer. In der ersten Variante der standard-html-renderkit.xml ist hierfür folgende Entity Deklaration zu finden:

```
<!ENTITY command-button-renderer-class 'com.sun.faces.renderkit.html_basic.ButtonRenderer'>
```

Die zweite Variante der `standard-html-renderkit.xml` hat wie die `faces-config.xml` die gewohnte Struktur von JSF Konfigurationsdateien nach der `web-facesconfig_1_1.dtd`. Das wichtige dieser Datei ist, dass hier alle HTML GUI Komponenten der JSF Referenzimplementierung mit ihren zugehörigen `Renderern` deklariert werden. Gemäß der `web-facesconfig_1_1.dtd` werden hierfür die `<component>` Elemente verwendet. Für den HTML Submit Button, bzw. `HtmlCommandButton` Tag sieht diese Deklaration folgendermaßen aus:

```
<faces-config>
  ...
  <component>
    ...
    <display-name>Command Button</display-name>
    <component-type>javax.faces.HtmlCommandButton</component-type>
    <component-class>javax.faces.component.html.HtmlCommandButton</component-class>

    &uicommand-props;
    &core-props;
    &events-props;
    &focus-props;
    &i18n-props;
    &input-props;
    &command-button-props;

    <component-extension>
      <base-component-type>javax.faces.Command</base-component-type>
      <renderer-type>javax.faces.Button</renderer-type>
    </component-extension>
    ...
  </component>

  <render-kit>
    ...
    <renderer>
      ...
      <component-family>javax.faces.Command</component-family>
      <renderer-type>javax.faces.Button</renderer-type>
      <renderer-class>&command-button-renderer-class;</renderer-class>

      &core-attrs;
      &events-attrs;
      &focus-attrs;
      &i18n-attrs;
      &input-attrs;
      &command-button-attrs;
    </renderer>
    ...
  <render-kit>
  ...
</faces-config>
```

Abbildung 7: Ausschnittt aus der `standard-html-renderkit.xml` für eine HTML Submit Schaltfläche. Was notwendig ist, damit alle HTML GUI Elemente für das JSF Framework verfügbar sind, wird nachfolgend dargestellt.

Es wird zuerst der vollklassifizierende Klassenname mit dem `<component-class>` Element des `HtmlCommandButton` festgelegt. Die Zuordnung des passenden `Renderers` erfolgt über den Aufruf der Entity beim Element `faces-config/render-kit/renderer/renderer-class`, die ja in der ersten Variante der `standard-html-renderkit.xml` definiert wurde. Nach diesem Strickmuster werden so alle HTML GUI Repräsentationsobjekte mit ihrem Renderer spezifiziert.

Durch den Aufruf von ConfigureListener.contextInitialized(ServletContext Event) werden auch diese Informationen in das FacesConfigBean überführt. Dies geschieht mittels der RenderKitRule, der RendererRule und der ComponentRule auf der einen und der Bean Objekte RenderKitBean, RendererBean und ComponentBean auf der anderen Seite. Im FacesConfigRuleSet wird für <component> Elemente dem Digester die ComponentRule hinzugefügt:

```
Digester.addRule("faces-config/component", new ComponentRule());
Digester.addCallMethod("faces-config/component/component-class", "setComponentClass", 0)
Digester.addCallMethod("faces-config/component/component-type", "setComponentType", 0)
```

Die ComponentRule erzeugt genau wie die NavigationRuleRule in Component Rule. begin(nameSpace, elementName, Attributes) ein ComponentBean Objekt und legt es mit Digester.push(ComponentBean) auf dem Stack des Digesters ab. Der Aufruf von ComponentRule.end(nameSpace, elementName) durch den Parseprozess des Digesters bewirkt, dass das ComponentBean mit FacesConfigBean.addComponent(ComponentBean) dem FacesConfigBean hinzugefügt wird.

Die CallMethodRule für die Elemente faces-config/component/component-class und faces-config/component/component-type rufen die Methoden ComponentBean.set ComponentClass(String) und ComponentBean.setComponentType(String) auf, nachdem sie das gegenwärtig oberste Digester Stackobjekt, das benötigte ComponentBean geholt hat, und übergibt mit dem String Parameter den Wert des component-type, bzw. des component-class Elements. Auf diese Weise kann später mittels ComponentBean.getComponentClass() das javax.faces.component.html.HtmlCommandButton Objekt instanziiert werden. Wo und wie dies geschieht, wird in den Ausführungen zum Request Processing Lifecycle in Szenario zwei behandelt.

Noch ist aber die Konfiguration der HTML GUI Komponente durch das ComponentBean nicht beendet. Es fehlt noch das entsprechende RendererBean für das ComponentBean. Die hierfür zuständige RendererRule arbeitet genau gleich wie die ComponentRule nur mit dem Unterschied, dass es sich nicht um ein ComponentBean dreht, sondern um ein RendererBean. D.h., Instanziierung eines RendererBean Objekts und dessen Ablage auf dem Stack des Digester in der Methode RendererRule.begin(nameSpace, elementName, Attributes). In ihr wird das derzeitig oberste Digester Stackobjekt, das RendererBean vom Stack gelöscht und dem mit FacesConfigBean.addRenderer(RendererBean) übergeben. Das Hinzufügen von CallMethodRules für die Elemente:

- faces-config/render-kit/renderer/component-family
- faces-config/render-kit/renderer/renderer-class
- faces-config/render-kit/renderer/renderer-type
- faces-config/render-kit/renderer/renderer-extension/renders-children
- faces-config/render-kit/renderer/renderer-extension/exclude-attributes
- faces-config/render-kit/renderer/renderer-extension/tag-name

sorgen für die Konfigurierung des jeweiligen RendererBean Objekts. Für das Setzen globaler Attribute für das ComponentBean und das RendererBean sorgen die Entity Aufrufe der <component> und der <renderer> Elemente, die im Listing grau markiert sind. Die aufgerufenen Entitäten verweisen auf xml Dateien, die sich im selben Verzeichnis befinden wie Variante zwei der standard-html-renderkit.xml. So wird z.B. mit dem Entitätenaufruf &command-button-attrs; auf die Deklaration <!ENTITY command-button-props SYSTEM "command-button-props.xml"> Bezug genommen. Im Element faces-config/render-kit/renderer wird der Inhalt der Datei command-button-props.xml mit in dieses Element eingebunden.

Wenn der Parser auf diese Entitäten stößt wandelt er die Entitätenaufrufe in den Inhalt der referenzierten xml Datei um.

Schritt 5: Das Parsen von Resourcen in /META-INF/

Es wird auch im Verzeichnis `META-INF` des Webapplikationshauptverzeichnisses, sofern vorhanden nach `faces-config.xml` Dateien gesucht:

```
Enumeration = Util.getCurrentLoader(ConfigureListener).getResources(META-INF/faces-config.xml)
```

In dieser `Enumeration` befinden sich durch einen `ClassLoader java.net.URL` Objekte, die auf den Pfad der Resourcedateien verweisen. Mittels dieses `URL`s kann der `Digester` angewiesen werden, auch die Inhalte diese Konfigurationsdateien in das `FacesConfigBean` zu übernehmen:

```
ConfigureListener.parse(Digester, URL, FacesConfigBean)
```

Eine weitere Möglichkeit JSF Konfigurationsinformationen zu hinterlegen ist das Verwenden eines `<init-param>` Parameters in der `web.xml` der Webapplikation. Unter einem `web.xml` Element `init-param/param-name` muss dort der Wert `javax.faces.CONFIG_FILES` und unter `init-param/param-value` eine kommaseparierter String mit JSF Konfigurationsdateien notiert werden, damit die Informationen dieser Resourcen in das `FacesConfigBean` übernommen werden können. Analog zum vorherigen Schritt wird zunächst der unter `init-param/param-value` eingetragene String anhand der Kommata gesplittet und mit `URL = ServletContext.getResource(dateiName)` in ein `URL` Objekt transformiert. Dieses kann dann wieder mit `ConfigureListener.parse(Digester, URL, FacesConfigBean)` dem `Digester` zur Übertragung in das `FacesConfigBean` übergeben werden.

Schritt 6: Das Parsen der /WEB-INF/faces-config.xml

Im vorletzten Schritt wird das Parsen der `/WEB-INF/faces-config.xml` gestartet, womit dann die Rule Objekte `ManagedBeanRule` und `NavigationRule` zu ihrem wichtigsten Einsatz kommen. Denn mindestens hier muss der Entwickler auf jeden Fall eine `faces-config.xml` pflegen, um dem JSF Framework zumindest das einwandfreie und fehlerfreie Hochfahren der Webapplikation zur ermöglichen. Mit den bereits bekannten Statements `URL = ServletContext.getResource(faces-config.xml)` und `ConfigureListener.parse(Digester, URL, FacesConfigBean)` werden die Informationen aller Elemente ebenfalls dem `FacesConfigBean` hinzugefügt.

Schritt 7: Überführung des FacesConfigBean in Laufzeitumgebung der Webapplikation

Da nun alle Konfigurationsdaten im `FacesConfigBean` zur Verfügung stehen, müssen diese entweder noch in das `Application` Objekt und/oder dessen Helferklasse `ApplicationAssociate` gebracht werden, oder in eine der Standardfabriken der JSF Distribution. Dies ist notwendig, damit die Informationen aus dem `FacesConfigBean` zur Laufzweit der Webanwendung bereitstehen. Das heisst genauer für das Request Processing, das nachdem die Webanwendung gestartet wurde, auf die erste Benutzeraktion wartet. Erfolgt diese, wird der Reqeust Processing Lifecycle durchlaufen und dieser bedient sich für seine Arbeit u.a. der Daten, die im `Application` und `ApplicationAssociate` Objekt oder den Fabrikklassen zu finden sind. Welche Daten dies sind, wie und wohin genau die Daten abgelegt werden, wird in den Ausführungen des letzten Schritts der Methode `ConfigureListener.contextInitialized(ServletContextEvent)` betrachtet.

In diesem letzten Schritt kommt die Methode `ConfigureListener.configure(ServletContext, FacesConfigBean, List)` zum Einsatz. Diese ruft je die Hilfsmethode für die folgenden Daten auf, die in `Application` und `ApplicationAssociate` Objekt und in den Fabriken zu speichern sind:

1. Das `LifecycleBean` für die Konfiguration eines `Lifecycle` Objekts. Andhand der vollklassfizierenden Klassennamens der `PhaseListener` Objekte des `LifecycleBeans` werden dem `Lifecycle` in dieser Methode die erzeugten `PhaseListener` mit `Lifecycle.addPhaseListener(PhaseListener)` hinzugefügt. Mittels `PhaseListener` Objekten hat der Applikationsentwickler die Möglichkeit, unmittelbar vor und nach dem Ausführen einer `Phase` des Request Processing Lifecycle programmiertechnisch einzugreifen.

   Die JSF Referenzimplementierung garantiert, das direkt vor der dem Ausführen einer `Phase` die Methode `PhaseListener.beforePhase(PhaseEvent)` aufgerufen wird und direkt nach Abschluss einer `Phase`, `PhaseListener.afterPhase(PhaseEvent)`. In diesen beiden `Phase` Callbackmethoden kann der Entwickler eigenen Code implementieren.

   Um diese Notifications über die `PhaseListener` Objekte zu erhalten, muss der Anwendungsentwickler in seiner `faces-config.xml` Einträge der folgenden Art haben:

```
<faces-config>
  ...
  <lifecycle>
    <phase-listener>
      one.two.three.Test1PhaseListener
    </phase-listener>
    <phase-listener>
      one.two.three.Test2PhaseListener
    </phase-listener>
    ...
  </lifecycle>
  ...
<faces-config>
```

2. Das `FactoryBean` für das Bekanntmachen aller Fabrikklassen des JSF Frameworks. Dazu gehören:
   - `com.sun.faces.application.ApplicationFactoryImpl`
   - `com.sun.faces.context.FacesContextFactoryImpl`
   - `com.sun.faces.lifecycle.LifecycleFactoryImpl`
   - `com.sun.faces.renderkit.RenderKitFactoryImpl`

   Mit den entsprechenden get Methoden des `FactoryBeans` wird der jeweilige vollklassifizierende Klassenname dem `FactoryFinder` mit `FactoryFinder.setFactory(String factoryName, String implName)` übergeben, wo der zweite Parameter unter dem Namen des ersten in einer `HashMap` geispeichert wird. Erzeugt wird an dieser Stelle noch keine Objektinstanz einer Fabrikklasse. Dies geschieht erst bei Aufruf der Methode `FactoryFinder.getFactory(String factoryName)`.

3. Das `ApplicationBean` für die initiale Erzeugung und Konfiguration eines `Application` Objekts. Die Erzeugung vollzieht sich in `ConfigureListener.application()`. Dort wird über den `FactoryFinder` eine `ApplicationFactory` instanziiert. In der `ApplicationFactory.getApplication()` Methode wird dann der Defaultkonstruktor der `Application` Implementierungsklasse aufgerufen der gleichzeitig auch die Helferklasse `ApplicationAssociate` mit `new ApplicationAssociate(Application)` erstellt.

   Die Konfiguration erfolgt über diverse set Methoden von `Application`. Folgende Objekte werden anhand der Informationen des `ApplicationBean` aus dem `FacesConfig`

Bean sprich letztlich aus der faces-config.xml des Entwicklers oder aus der des JSF Frameworks, sprich der jsf-ri-config.xml, dem Application Objekt übergeben.

- ActionListenerImpl
- NavigationHandlerImpl
- PropertyResolverImpl
- StateManagerImpl
- VariableResolverImpl
- ViewHandlerImpl
- Der vollklassifizierende Klassenname von com.sun.faces.renderkit.RenderKit Impl

Die Instanziierung dieser Objekte übernimmt die Utilityklasse Object = Util.create Instance(String, Class, Object). Der String Parameter ist der vollklassifizierende Klassenname des zu erzeugenden Objekts, der zweite ist die java.lang.Class Variante des Zielobjekts, die mit <className>.class angegeben wird. Der Object Parameter ist bei allen Aufrufen dieser Utilitymethode immer null, weil Application.getXYZ() immer null ergibt. Erst im nächsten Statement wird die entsprechende Application.setXYZ(XYZ) aufgerufen, die dann das erzeugte Objekt von Util.createInstance(String, Class, Object) übergeben bekommt.

4. ComponentBean Objekte für die Überführung aller HTML GUI Elementobjekte mittels Application.addComponent(String, String) durch die Methode Configure Listener.configure(ComponentBean[]) in das Application Objekt. Die Werte der beiden genannten xml Elemente aller JSF HTML GUI Elemente werden in Configure Listener.configure(ComponentBean[]) mit der Methode Application.add Component(String, String) dem Application Objekt verfügbar gemacht. Die beiden String Parameter enthalten das jeweilige component-type und component-class Element des jeweiligen <component> Superelements. Auch hier werden keine UIComponent Derivate erzeugt. Dies passiert erst während des Request Processing Lifecycle.

5. ManagedBeanBean Objekte für die <managed-bean> Elemente der faces-config.xml, für die der Applikationsentwickler verantwortlich ist. In ConfigureListener.configure(ManagedBeanBean[]) werden aus allen ManagedBeanBeans, die der Digester erstellt hat, ManagedBeanFactory Objekte erzeugt, die in ihrem Konstruktor ein ManagedBeanBean aus dem übergebenen Array zur Initialisierung erhalten.

   Dieses ManagedBeanFactory Objekt wird dann dem ApplicationAssociate mit ApplicationAssociate.addManagedBeanFactory(String, ManagedBeanFactory) unter dem vom Entwickler gewählten Namen aus dem faces-config/managed-bean /managed-bean-name Element in einer HashMap für die Laufzeit der Webanwendung gesichert, also für den Request Processing Lifecycle verfügbar gemacht. Nach Ende der Abarbeitung der Methode ConfigureListener.configure(ManagedBeanBean[]) sind alle Informationen der faces-config/managed-bean Elemente in der dafür vorgesehenen HashMap des ApplicationAssociate Objekts vorhanden.

6. NavigationRuleBean Objekte für die Speicherung der Inhalte aller <navigation-case> Elemente mit ihrem dazugehörenden <from-view-id> Element im Application Associate Objekt, die sich in der faces-config.xml der konkreten Webanwendung befinden. Das gegenwärtige NavigationRuleBean selbst wird nicht gespeichert. Jeder navigation case wird einzeln in einem ConfigNavigationCase Objekt mittels Application Associate.addNavigationCase(ConfigNavigationCase) gesichert und nicht in der Struktur, wie sie in der faces-config.xml vorkommt. Übernommen wird aus jedem <navigation-case> Element der Inhalt der Elemente <from-outcome>, <to-view-id>, und <from-view-id>. Dies passiert mit entsprechenden set Methoden von Config

NavigationCase. Z.B. ConfigNavigationCase.setFromViewId(NavigationRuleBean.getFromViewId()) für ein <from-view-id> Element eines <navigation-case>.

7. RenderKitBean Objekte für die Erzeugung und Verfügbarkeit eines RenderKits für den Request Processing Lifecycle in der RenderKitFactoryImpl der Referenzimplementierung. Der Sinn des RenderKits besteht in Verwaltung, sprich dem Hinzufügen und Beziehen von bereits hinzugefügten Renderer Objekten für HTML GUI Elemente. In dieser Funktion kann das RenderKit als eine gewöhnliche Fabrikklasse angesehen werden.

   Das Registrieren aller Renderer Klassen für alle HTML GUI Elemente, die die JSF Referenzimplementierung unterstützt, geschieht in drei Schritten:
   1. Das Holen einer RenderKitFactory mit (RenderKitFactory)FactoryFinder.getFactory(FactoryFinder.RENDER_KIT_FACTORY). Falls noch nicht initialisiert, erledigt dies der FactoryFinder und sichert sie in seinem Fabrikobjekt Pool, seiner HashMap.
   2. Das Holen eines RenderKits über die gewonnene RenderKitFactory mit RenderKitFactory.getRenderKit(null,RenderKitBean.getRenderKitId()). Falls dies null ergibt muss das RenderKit anhand des vollklassifizierenden Klassen-namens von RenderKitBean.getRenderKitClass() und anschließend mit einem ClassLoader instanziiert werden. Nun kann das gerade instanziierte RenderKit der RenderKitFactory mit RenderKitFactory.addRenderKit(RenderKitBean.getRenderKitId()) hinzugefügt werden und steht somit dem request processing mit RenderKitFactory.getRenderKit(FacesContext, renderKitId) zur Verfügung.
   3. Im letzten Schritt müssen dem RenderKit noch die Renderer des Render KitBean bekannt gemacht werden. Hierfür gibt es die private Hilfsmethode ConfigureListener.configure(RendererBean[], RenderKit). Zunächst wird analog zum RenderKit jedes Objekt anhand des vollklassifizierenden Klassen-namens aus RenderKitBean.getRendererClass() geladen, sprich Renderer = Util.getCurrentLoader(this).loadClass(RenderKitBean.getRendererClass()).newInstance(). Mit dem letzten Statement wird der gewonnene Renderer dem RenderKit hinzugefügt: RenderKit.addRenderer(RenderKitBean.getComponentFamily(), RenderKitBean.getRendererType(), Renderer).

   Ergebnis: Es steht für den Request Processing Lifecycle nun jeder Renderer für jedes HTML GUI Element bereit. Mit den Callbackmethoden Renderer.decode(FacesContext, UIComponent), Renderer.encodeBegin(FacesContext,UIComponent), Renderer.encodeChildren(FacesContext,UIComponent) und Renderer.encodeEnd(FacesContext,UIComponent) des jeweiligen Renderer kann nun HTML Code jür die JSP Seiten erzeugt und in die andere Richtung ausgewertet werden.

8. ValidatorBean Objekte für die Überführung der Inhalte der Elemente faces-config/validator der faces-config.xml oder anderen xml Resource Dateien mit einem faces-config Wurzelelement. So definiert die JSF Referenzimplementierung in der Datei jsf-ri-config.xml des Verzeichnisses com.sun.faces.util folgende Standardvalidatoren:
   - javax.faces.validator.DoubleRangeValidator
   - javax.faces.validator.LengthValidator
   - javax.faces.validator.LongRangeValidator

   Dem Application Objekt werden mit Application.addValidator(String, String) diese Validatoren hinzugefügt. Wie in Schritt 4 werden hier nicht die instanziierten Objekte gespeichert sondern die nur die vollklassifizierenden Klassennamen. Erzeugt werden die

Instanzen erst beim Aufruf der `Application.createValidator(String)` Methode, indem anhand des String Parameters der Klassenname aus der Validator `HashMap` bezogen wird und damit die Instanz erzeugt werden kann.

Letzter Akt: `FacesServlet.init(ServletConfig)`

Der Aufruf der `FacesServlet.init(ServletConfig)` Methode nachdem alle Konfigurationsresourcen ausgelesen, in das `FacesConfigBean` gespeichert, und in die globalen Frameworkobjekte überführt wurden, bildet den letzten Akt des Hochfahrens der JSF Webapplikation(en). Im Gegensatz zur `init` Methode des `ActionServlets` des Struts Frameworks geschieht beim `FacesServlet` vergleichsweise wenig.

Es werden lediglich drei Objekte instanziiert, die für den Request Processing Lifecycle von Bedeutung sind: die `FacesContextFactoryImpl`, das `ApplicationImpl` und die `Lifecycle Impl`.

2.4 Das Request Processing

Szenario zwei befasst sich mit den Schritten, die beim Request Processing Lifecycle durchlaufen werden, um nach einer vom Benutzer ausgelösten Aktion die richtige HTML Seite als Response anzuzeigen. Der Request Processing Lifecycle ist das Herzstück des Request Handlings. Je nach Komplexität des Requests, sprich ob z.B. Datenbankzugriffe und Datenbarbeitungsroutinen ausgeführt werden müssen, durchläuft der Request alle Phasen des Lifecycles. Geht es beim Request aber nur um einen Reload der gegenwärtigen Seite, ist die Response nach dem Durchlauf von zwei Phasen bereit zur Anzeige.

Bei jeder Benutzeraktion, also nach dem Klick auf eine Schaltfläche oder auf einen Link, oder aber auch nach dem Auslösen eines Ergeignisses wie dem Auswählen eines Listenelements beginnt der Durchlauf des Request Processing Lifecycles mit `FacesServlet.service(Servlet Request, ServletResponse)` durch den Tomcat Webserver. Anhand eines Beispiels in dem der Benutzer Namen und Passwort zur Anmeldung für eine Webseite benötigt, wird die Implementierung des kompletten Lifecycles, sprich aller Phasen, demonstiert.

Anhand der Implementierung der `FacesServlet.service(Servlet Request, Servlet Response)` Methode wird durch den Aufruf der zwei Methoden `Lifecycle.execute(Faces Context)` und `Lifecycle.render(FacesContext)` deutlich, dass der Request Processing Lifecycle in zwei Teile zerlegt werden kann: der erste Teil, `Lifecycle.execute(Faces Context)`, besteht aus den Phasen eins bis fünf, der zweite `Lifecycle.render(Faces Context)`, dient dazu den HTML Code zu generieren, auf Basis der Daten die in den Phasen zuvor in den globalen Framework Objekten gespeichert wurden. Sprich in das `Application` Objekt, dem `FacesContext` und dem `ExternalContext`.

Schritt 1: `Lifecycle.execute(FacesContext)`

Das folgende Codelisting zeigt den Aufruf des `FacesServlets`, nachdem der Benutzer auf die Schaltfläche zum Absenden der Anmeldedaten, geklickt hat.

```
at com.sun.faces.application.ActionListenerImpl.processAction(ActionListenerImpl.java:78)
at javax.faces.component.UICommand.broadcast(UICommand.java:312)
at javax.faces.component.UIViewRoot.broadcastEvents(UIViewRoot.java:266)
at javax.faces.component.UIViewRoot.processApplication(UIViewRoot.java:380)
at com.sun.faces.lifecycle.InvokeApplicationPhase.execute(InvokeApplicationPhase.java:75)
at com.sun.faces.lifecycle.LifecycleImpl.phase(LifecycleImpl.java:200)   // 3.
at com.sun.faces.lifecycle.LifecycleImpl.execute(LifecycleImpl.java:90)  // 2.
at javax.faces.webapp.FacesServlet.service(FacesServlet.java:197)        // 1.
at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:284)
at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:204)
at org.apache.catalina.core.StandardWrapperValve.invoke(StandardWrapperValve.java:257)
at org.apache.catalina.core.StandardValveContext.invokeNext(StandardValveContext.java:151)
at org.apache.catalina.core.StandardPipeline.invoke(StandardPipeline.java:567)
at org.apache.catalina.core.StandardContextValve.invokeInternal(StandardContextValve.java:245)
at org.apache.catalina.core.StandardContextValve.invoke(StandardContextValve.java:199)
at org.apache.catalina.core.StandardValveContext.invokeNext(StandardValveContext.java:151)
at org.apache.catalina.core.StandardPipeline.invoke(StandardPipeline.java:567)
at org.apache.catalina.core.StandardHostValve.invoke(StandardHostValve.java:184)
at org.apache.catalina.core.StandardValveContext.invokeNext(StandardValveContext.java:151)
at org.apache.catalina.valves.ErrorReportValve.invoke(ErrorReportValve.java:164)
at org.apache.catalina.core.StandardValveContext.invokeNext(StandardValveContext.java:149)
at org.apache.catalina.core.StandardPipeline.invoke(StandardPipeline.java:567)
at org.apache.catalina.core.StandardEngineValve.invoke(StandardEngineValve.java:156)
at org.apache.catalina.core.StandardValveContext.invokeNext(StandardValveContext.java:151)
at org.apache.catalina.core.StandardPipeline.invoke(StandardPipeline.java:567)
at org.apache.catalina.core.ContainerBase.invoke(ContainerBase.java:972)
at org.apache.coyote.tomcat5.CoyoteAdapter.service(CoyoteAdapter.java:206)
at org.apache.coyote.http11.Http11Processor.process(Http11Processor.java:833)
at org.apache.coyote.http11.Http11Protocol$Http11ConnectionHandler.processConnection(Http11Protocol.java:732)
at org.apache.tomcat.util.net.TcpWorkerThread.runIt(PoolTcpEndpoint.java:619)
at org.apache.tomcat.util.threads.ThreadPool$ControlRunnable.run(ThreadPool.java:688)
at java.lang.Thread.run(Unknown Source)
```

Abbildung 8: Codelisting über das Starten des JSF Request Processing Lifecycle durch den Tomcat Webserver, der Ausgangspunkt für alle direkt folgenden Untersuchungen.

Durch den Aufruf von `Lifecycle.execute(FacesContext)` werden in einer Schleife alle Phasen, sprich `com.sun.faces.lifecycle.Phase` Derivate mit Ausnahme der `Render ResponsePhase` ausgeführt. Hierfür existiert im `LifecycleImpl` Objekt ein globaler `Phase[]` Array der mit folgendem Inhalt gefüllt ist:

1. mit `null` als Platzhalter für die AnyPhase Phase, die nicht Bestandteil des Request Processing Lifecycle ist und für die auch keine Implementierungsklasse existiert.
2. mit den `Phase` Derivaten: `RestoreViewPhase`, `ApplyRequestValuesPhase`, `Process ValidationsPhase`, `UpdateModelValuesPhase` und der `InvokeApplicationPhase`.

Das Ausführen einer `Phase` besteht aus drei Teilen. Erledigt werden diese Schritte in der privaten Hilfsmethode `LifecycleImpl.phase(PhaseId, Phase, FacesContext)`, die von `LifecycleImpl.execute(FacesContext)` aufgerufen wird:

1. Das Benachrichtigen aller `PhaseListener` Objekte, direkt **vor** dem Ausführen einer `Phase`. Dazu wird zunächst mit `new PhaseEvent(FacesContext, PhaseId, LifecycleImpl)` ein `PhaseEvent` erzeugt. Dies wird dem `PhaseListener` durch seine Callbackmethode `PhaseListener.beforePhase(PhaseEvent)` zugewiesen.
2. Das Ausführen der `Phase` Derivate aus 2. s.o. Dazu wird `Phase.execute(FacesContext)` ausgeführt. Hinter dem `Phase` Objekt verbirgt sich durch die Polymorphie immer das gegenwärtige konkrete `Phase` Derivat.

3. Das Benachrichtigen aller `PhaseListener` Objekte, direkt **nach** dem Ausführen einer `Phase`. Es wird wie bei 1. ein `PhaseEvent` erstellt und dem `PhaseListener` in der zweiten Callbackmethode `PhaseListener.afterPhase(PhaseEvent)` übergeben. Die `Phase Listener` hat der Entwickler in der `faces-config.xml` deklariert, siehe Kapitel 2.3 Schritt sieben.

Um die Implementierungsdetails des Ablaufs des Request Processing Lifecycles zu untersuchen, wird ein Standardbeispiel herangezogen, das sehr häufig in der Praxis von Webanwendungen auftaucht: Es handelt sich um die Eingabe von Benutzername –und passwort beim Einloggen auf die Haupseite einer Website. Die folgenden beiden Listings zeigen den JSF HTML Code der Login Seite, sowie den Ausschnitt der dazugehörigen Einträge in der `faces-config.xml`.

```
<%@ taglib uri="/WEB-INF/html_basic.tld" prefix="h" %>
<%@ taglib uri="/WEB-INF/jsf_core.tld" prefix="f" %>

<f:view>
  <h:form>
    <h:panelGrid columns="2">
       <h:outputText value="User Name:" />
       <h:inputText size="30" value="#{loginBean.userName}" />

       <h:outputText value="Password:" />
       <h:inputText size="30" value="#{loginBean.password}"/>
    </h:panelGrid>

    <h:panelGrid columns="1">
      <h:commandButton action="#{loginBean.invokeAction}" value="Login"/>
    </h:panelGrid>
  </h:form>
</f:view>
```

Abbildung 9: JSP Codeausschnitt mit Name und Passwort Textfeldern und Submit Schaltfläche.

```
<faces-config>
  ...
  <navigation-rule>
    <from-view-id>/view/user/login.jsp</from-view-id>
    <navigation-case>
      <from-outcome>success</from-outcome>
      <to-view-id>/view/main/main.jsp</to-view-id>
    </navigation-case>
  </navigation-rule>

  <managed-bean>
    <managed-bean-name>loginBean</managed-bean-name>
    <managed-bean-class>com.tsystems.jsf.user.bean.UserLoginBean</managed-bean-class>
    <managed-bean-scope>request</managed-bean-scope>
  </managed-bean>
  ...
<faces-config>
```

Abbildung 10: Zugehöriger `faces-config.xml` Abschnitt für die JSP aus Abbildung 8 und das Beanobjekt als Modelklasse.

Bevor der Request Procsssing Lifecycyle durchlaufen werden kann, muss zuerst der JSF Code der jeweiligen gegenwärtigen Seite dem Framework zugänglich gemacht werden. Das bedeutet, dass die Tags und deren Attributsinhalte in die von der JSP vorgegebenen Baumstruktur in die JSF GUI Elementrepräsentationsobjekte der Pakete `javax.faces.component` und `javax.`

faces.component.html überführt werden müssen. Den Vorgang von der Transferierung von JSF GUI Element Tags in deren Javarepräsentationsobjekte nennt man **Encoding**. Wie dies realisert ist, wird in Schritt 1.1 analysiert.

Schritt 1.1: Das Encoding des JSF JSP Codes

Grundlage für das Encoding jeder JSP Seite ist eine vom JSP Compiler des Tomcat Webservers generierte Servlet Klasse. So ist z.B. der JSF HTML Code des Listings aus der JSP loginBody.jsp entnommen. Der JSP Compiler generiert daraus die Javaklasse loginBody_jsp.java. Für jedes JSF JSP GUI Element gibt es ein Tag Repräsentationsjavaobjekt. Deklariert werden diese Objekte in den tld Dateien html_basic.tld und jsf_core.tld, die gewöhnlich im WEB-INF Ordner unter dem Webapplikationshauptverzeichnis zu finden sind. Für die in diesem Beispiel vorkommenden Tags aus dem Paket com.sun.faces.taglib.html_basic und com.sun.faces.taglib.jsf_core sind dies:

```
f:view             : com.sun.faces.taglib.jsf_core.ViewTag
h:form             : com.sun.faces.taglib.html_basic.FormTag
h:panelGrid        : com.sun.faces.taglib.html_basic.PanelGridTag
h:outputText       : com.sun.faces.taglib.html_basic.OutputTextTag
h:inputText        : com.sun.faces.taglib.html_basic.InputTextTag
h:commandButton    : com.sun.faces.taglib.html_basic.CommandButtonTag
```

Für die Instanziierung und den Aufbau dieser Objekte mit ihrem Elternelement und ihren evtl. Kindelementen existiert in der Klasse loginBody_jsp.java jeweils eine entsprechende Methode. Das generelle Ziel besteht nun darin, mittels der Klasse loginBody_jsp.java, ein ViewTag Objekt zu erstellen, von aus man auf alle Unterelemente, sprich JSF Tag Repräsentationsobjekte aus obigen Liste traversieren kann.

Die Methode loginBody_jsp._jspx_meth_f_**view**_0(PageContext) ist der Startpunkt. Hier wird zuerst über den org.apache.jasper.runtime.TagHandlerPool ein ViewTag Objekt bezogen. Da der ViewTag als einziges Element kein Elternelement haben kann, folgt das Statement ViewTag.setParent(null). Als nächstes sind die Standardmethoden an der Reihe, die jedes Tag Derivat implemtieren muss. Die wichtigsten sind: Tag.doStartTag(), Tag.doAfterBody(), und Tag.doEndTag().

In ViewTag.doStartTag() wird zunächst die gleiche Methode der Basisklasse aufgerufen: UIComponentBodyTag.doStartTag(). Da diese nicht in dieser Klasse zu finden ist, wird deren Basisklasse weitergeleitet: UIComponentTag.doStartTag(). Was in dieser Methode passiert wird am Ende des Schrittes 1.1 erläutert. Was ebenfalls am Ende dieses Schrittes zu klären ist, ist die Speicherung des Zustandes der loginBody.jsp. Das Objekt, das den Zustand repräsentiert, wurde im Abschnitt 2.2.1 unter javax.faces.application.StateManager erläutert.

Mit dem FacesContext wird im nächsten Schritt ein com.sun.faces.renderkit.html_basic.HtmlResponseWriter bezogen, um dessen Methode HtmlResponseWriter.startDocument() aufzurufen, die keine Implementierung enthält. Am Ende von ViewTag.doStartTag() wird im Erfolgsfall der Wert der Integerkonstanten javax.servlet.jsp.tagext.Tag.EVAL_BODY_INCLUDE zurückgegeben und es geht weiter in loginBody_jsp.java.

Direkt nach dem <f:view> Element folgt in der loginBody.jsp ein <h:form>. Dies wird als das gewöhnliche HTML <form> Element implementiert. In der compilierten Javaklasse der Beispiel JSF JSP wurde eine diesem Element entsprechende Methode loginBody_jsp._jspx_meth_h_**form**_0(ViewTag, PageContext) generiert, die nun zum Aufruf kommt. Hier

wird die Implementierung der JSF Elementhierarchie der loginBody.jsp deutlich, die nach diesem Muster in allen JSF JSP Seiten gleich realisiert wird. Jede javax.servlet.jsp.tagext.Tag Instanziierungsmethode in der generierten Javaklasse einer JSP muss von der Instanziierungsmethode des Elternelements aufgerufen werden. Daraus folgt hier: loginBody_jsp._jspx_meth_f_**view**_0(PageContext) ruft loginBody_jsp._jspx_meth_h_**form**_0(ViewTag, PageContext) auf.

Analog zu der eben besprochenen ViewTag Methode wird in loginBody_jsp._jspx_meth_h_**form**_0(ViewTag,PageContext) zuerst ein com.sun.faces.taglib.html_basic.FormTag Objekt vom TagHandlerPool bezogen. Diesem wird der PageContext mit FormTag.setPageContext(PageContext) übertragen. Im Unterschied zum ViewTag besitzt der FormTag ein Elternelement gemäß der Hierarchie der JSF Elemente der loginBody.jsp. Deshalb lautet der Parameter beim Aufruf von FormTag.setParent(**ViewTag**) und nicht null. Nun kommt FormTag.doStartTag() an die Reihe, um das Rendern des HTML <form> Elements einzuleiten. Diese Methode sieht außer dem Auruf derselben Methode der Superklasse UIComponentTag.doStartTag() keine Implementierung vor. Gleiches gilt analog für die FormTag.doEndTag() Methode.

Von Bedeutung sind in loginBody_jsp._jspx_meth_h_form_0(ViewTag, PageContext) die Aufrufe der Methoden für die beiden <h:panelGrid> Elemente. Die Aufgabe der Methode _jspx_meth_h_panelGrid_0(javax.servlet.jsp.tagext.JspTag,PageContext) für das erste <h:panelGrid> Element ist die Encodierung zweier <h:outputText> und <h:input Text> Elemente für die Namens- und Passworteingabe. Der JspTag Parameter ist das Elternelement, in diesem Fall der FormTag. Die Encodierung des zweiten <h:panelGrid> Elements über _jspx_meth_h_panelGrid_1(javax.servlet.jsp.tagext.JspTag, PageContext) beinhaltet ein einzelnes <h:commandButton> Element.

In loginBody_jsp._jspx_meth_h_panelGrid_0(JspTag, PageContext) werden zu Beginn allgemeine Tag Daten gesetzt:

- PanelGridTag.setPageContext(PageContext)
- PanelGridTag.setParent(FormTag)
- PanelGridTag.setColumns("2")
- PanelGridTag.setStyleClass("normal")

Die beiden letzten Aufrufe nehmen ihre Parameterwerte dirket aus den Attibuten columns und styleClass der JSP. Das styleClass Attribut und dessen Wert stammen aus einer Style Sheet Einbindung einer Master JSP in die jede andere JSP dieser Webapplikation eingebettet ist. Für die beiden <h:outputText> und <h:inputText> Elemente werden die Methoden loginBody_jsp._jspx_meth_h_**outputText**_x(JspTag, PageContext) und loginBody_jsp._jspx_meth_h_**inputText**_x(JspTag, PageContext) aufgerufen. Das x steht für die laufende Nummer des entsprechenden Elements. Hinter dem Parameter JspTag steht der PanelGridTag.

Beim Aufruf der Methoden für das <h:outputText> Element passiert fast nichts Neues. Nach dem Setzen der gleichen Attribute, wie es auch bei den bereits besprochenen JspTags der Fall war, folgt der Aufruf der Methoden OutputTextTag.doStartTag() und OutputTextTag.doEndTag(). Beiden ist gemeinsam, wie bei allen Tags ausser denen des ViewTags, dass lediglich dieselbe Methode der Basisklasse UIComponentTag.doStartTag(), bzw. UIComponentTag.doEndTag() zur Ausführung kommen. Einzig zu erwähnen bleibt das Setzen value binding Ausdrucks OutputTextTag.setValue("#{beanName.attributeName}") unter dem value Attribut von <h:outputText>. Bei diesem JSF Element kann es aber auch sein, dass unter dem value Attribut ein einfacher Text zur simplen Anzeige hinterlegt ist. Für die Phasen

ApplyRequestValuesPhase und UpdateUserModelPhase sind value binding Ausdrücke von zentraler Bedeutung.

Bei den Methoden für <h:inputText> Elemente ist ebenfalls neben dem Setzen von Standardattributen von InputTextTag das value binding wichtig. Hinter dem value Attribut dieses JSF JSP Elements muss zwingend ein value binding Ausdruck stehen, da bei einem Textfeld immer der Wert eines Beanklassenattributs dargestellt wird. Also muss in loginBody_jsp._jspx_meth_h_inputText_0(JspTag, PageContext) folgendes Attribut gesetzt sein: InputTextTag.setValue("#{beanName.attributeName}").

Da die beiden <h:outputText> und <h:input Text> Elemente keine Kindelemente haben können, ist hier in der loginBody.jsp die tiefste Hierarchiestufe erreicht. Es fehlt noch die Funktionalität, die hinter dem <h:commandButton> Element steht. Dieses ist in diesem Beispiel auch in einem <h:panelGrid> Element enthalten. Die dem <h:commandButton> Element entsprechende Methode _jspx_meth_h_**commandButton**_0(JspTag, PageContext) innerhalb der Klasse loginBody_jsp wird von loginBody_jsp._jspx_meth_h_panelGrid_1(JspTag, PageContext) aus aufgerufen.

Nach dem Erhalten eines CommandButtonTag Objekts über einen TagHandlerPool, sind die beiden wichtigsten Anweisungen das Setzen der Werte aus den Attributen action und value des <h:commandButton> Elements. Es handelt sich dabei um eine value binding Expression, die im Falle des value Attributs auf die Methode public String invokeAction() des Beans com.tsystems.jsf.user.bean.UserLoginBean verweist. Dieser Wert wird mit CommandButtonTag.setAction("#{loginBean.invokeAction}") übergeben. Der Text, der auf der Schaltfläche steht, wird mit CommandButtonTag.setValue("Login") gesetzt. Die beiden Callbackmethoden CommandButtonTag.doStartTag() und CommandButtonTag.doEndTag() beinhalten dieselbe Funktionalität wie alle anderen Tags ausser dem ViewTag.

Zwischenstand: Die JSF Elementhierarchiestruktur wurde in ein ViewTag Objekt gespeichert, das alle Unterelemente und deren Unterelemente in sich vereint. Erreicht wurde dies durch die rekursiven Methodenaufrufe der jeweiligen korrespondierenden JSF JSP Elemente. Nun muss das ViewTag Objekt für die Phasen des Request Processing Lifecycle gesichert und zugänglich gemacht werden, damit die gesamte loginBody.jsp für die weitere Requestbearbeitung ausgewertet werden kann. Angestoßen wird dies in der Callbackmethode ViewTag.doAfterBody().

Noch nicht betrachtet wurde die Funktionalität der Methode UIComponentTag.doStartTag(). In dieser Methode geht es darum, den HTML Code für das gegenwärtige UIComponentTag Objekt zu generieren. Dazu müssen folgende Schritte erledigt werden:

1. Das Konfigurieren einer javax.faces.context.ResponseWriter Implementierung und deren Speicherung in den FacesContext.
2. Das Holen oder Erzeugen eines dem gegenwärtigen UIComponentTag entsprechenden UIComponent Objekts. Z.B. für den CommandButtonTag ist dies der HtmlCommandButton.
3. Das Speichern dieses UIComponent Objekts als Kind des Eltern UIComponent Objekts.
4. Das Umwandeln der UIComponent Objekthierarchie unter UIViewRoot in HTML Code mittels der Methoden UIComponentTag.encodeBegin(), UIComponentTag.encodeChildren(), und UIComponentTag.encodeEnd() für die Statusspeicherung der JSF JSP unter einem <h:hidden> Element.

Ein ResponseWriter wird in Form eines HtmlResponseWriters in der Methode UIComponentTag.setupResponseWriter() mit Hilfe der RenderKitImpl instanziiert. Der Methode RenderKitImpl.createResponseWriter(java.io.Writer, contentType, encoding) wird als erster Parameter ein Writer übergeben, der als inner class erzeugt wird. Das

encoding ist standardmäßig ISO-8859-1, der content type ist text/html. Die vielen Writer.write(...) Methoden benutzen den javax.servlet.jsp.JspWriter des Page Contexts, um darin den HTML Code im String Format zu schreiben. In dieser RenderKit Impl Methode wird mittels einfachem Konstruktoraufruf ein HtmlResponseWriter erzeugt, dem der per inner class erzeugten Writer mitgeliefert wird. Nach der Erzeugungsprozedur wird der HtmlResponseWriter in UIComponentTag.setupResponseWriter() im FacesContext mit FacesContext.setResponseWriter(ResponseWriter) gesichert.

Das Beschaffen des UIComponent Objekts, das dem UIComponentTag enspricht, vollzieht sich in zwei Schritten. Zuerst wird der Elterntag des gegenwärtigen UIComponentTag geholt. Zweitens wird durch die Methode UIComponentTag.findComponent(PageContext) das dem UIComponentTag entsprechende UIComponent erzeugt. Der Kern der Funktionalität spielt sich in der zuletzt genannten Methode ab. Am Ende der UIComponentTag.doStartTag() Methode wird jeder Tag mit einem Stack ähnlichen Mechnismus gespeichert.

Realisiert wird dieser durch die Methoden UIComponentTag.pushUIComponentTag() und UIComponentTag.popUIComponentTag(). Mit Hilfe einer java.util.List die im Page Context des UIComponentTag unter dem Attribut javax.faces.webapp.COMPONENT_TAG_ STACK gespeichert wird, wird jeder UIComponentTag der List in UIComponentTag.pushUI ComponentTag() hinzugefügt. Die so um ein Element erweiterte List kommt dann wieder im aktualisierten Zustand zurück mit PageContext.setAttribute(javax.faces.webapp. COMPONENT_TAG_STACK, List, PageContext.REQUEST_SCOPE) in den PageContext. Um den neuesten auf diesem Stack abgelegten UIComponentTag wieder zu entfernen, wird in UIComponentTag.popUIComponentTag() mit List.remove(List.size() - 1) immer das letzte Element gelöscht und anschließend die List wieder in den PageContext übertragen, genau wie in UIComponentTag.pushUIComponentTag().

Um nun den Elterntag des gegenwärtigen UIComponentTag zu holen, wird die Methode UIComponentTag.getParentUIComponentTag() aufgerufen, die genauso arbeitet wie UIComponentTag.popUIComponentTag(), nur dass das erhaltene UIComponentTag Objekt nicht gelöscht, sondern zurückgegeben wird.

Der Aufruf der Methode UIComponentTag.findComponent(FacesContext) leitet das Erstellen des gegenwärtigen UIComponent Objekts ein. Nach dem Ermitteln des Eltern UIComponentTags wird, sofern dieser nicht null ist, mit UIComponentTag.getComponent Instance() dessen entsprechende UIComponent Version besorgt. Diese ist sofort verfügbar, weil für das Eltern UIComponentTag Objekt ja schon der UIComponent Erzeugungsprozess abgeschlossen wurde. Ergibt UIComponentTag.getParentUIComponentTag(PageContext) null, handelt es sich beim gegenwärtigen UIComponent Objekt um UIViewRoot, bzw. beim ensprechenden UIComponentTag Objekt um den ViewTag. In diesem Fall muss UIViewRoot in den PageContext gespeichert werden: PageContext.setAttribute(javax.faces.webapp. CURRENT_VIEW_ROOT, UIViewRoot, PageContext.REQUEST_SCOPE). Mit der Rückgabe des UIViewRoot Objekts wird frühzeitig in die Methode UIComponentTag.doStartTag() zurückgekehrt.

Bei jedem anderen Tag als dem ViewTag, wird für UIComponentTag.getParent UIComponentTag(PageContext) nicht null zurückgeliefert. In diesem Fall muss für das gegenwärtige UIComponentTag Objekt das entsprechende UIComponent neu geschaffen werden. Dazu wird zuerst ein für jedes UIComponent eine notwendige eindeutige id generiert. Diese Funktionalität ist der Methode UIViewRoot().createUniqueId() vorbehalten. Diese zu generierende id besteht aus dem Prefix _id und daran gehängt eine laufende Nummer.

Mit der Erstellung der Facets für das Eltern UIComponent Objekt beschäftigt sich der nächste Schritt. Da dieser aber für das einfache Beispiel aus den vorangegangenen Listings nicht relevant ist, wird der eigentliche Instanziierungsprozess für das UIComponent Objekt betrachtet. Eingeleitet und an das Application Objekt delegiert wird dieser durch die Methode UIComponent.createChild(FacesContext, UIComponent, componentId). Diese ruft eine weitere Hilfsmethode UIComponent.createComponent(FacesContext, componentId) die letztendlich die Delegation an das Application Objekt mit Application.createComponent(componentType) durchführt. Diese Methode gibt es noch in einer anderen Variante: Application.createComponent(ValueBinding, FacesContext, componentType). Diese wird statt der erst genannten verwendet, wenn in der JSP für das dem UIComponent, bzw. dem UIComponentTag entsprechenden JSF JSP Element das Attribt binding gesetzt wurde. Da dies im Beispiel nicht vorkommt gilt die Aufmerksamkeit Application.createComponent(componentType). Der Wert des componentType Parameters wird durch durch die Methode UIComponentTag.getComponentType() geliefert. Konkrete Ableitungen von UIComponent Tag wie CommandButtonTag geben Identifier zurück wie javax.faces.HtmlCommandButton.

Diese entsprechen den Werten unter den Elementen faces-config/component/component-type aus der standard-html-renderkit.xml. Die Methode Application.createComponent(componentType) erstellt mit der Hilfsmethode ApplicationImpl.new Thing(Object,Map) eine Instanz eines UIComponent Objekts. Hinter dem Object Parameter steht der componentType des konkreten UIComponentTag. Map beinhaltet alle Werte der faces-config/component/component-class Elemente der standard-html-renderkit.xml, die während des Frameworkinitialisierungsprozesses über das FacesConfigBean durch die Methode FacesConfigListener.configure(ComponentBean[]) mit ApplicationImpl.addComponent(componentType, componentClass) in die HashMap ApplicationImpl.componentMap gelangt sind.

Die Instanziierung des konkreten UIComponent Objekts erfogt durch die folgenden wesentlichen Schritte:

1. Bezug des vollklassifizierenden Klassennamens des UIComponent Objekts mit Object = Map.get(Object). Der Object Parameter stammt aus dem Methodenparameter ApplicationImpl.newThing(Object,Map). Der Rückgabewert Object ist der vollklassifizierende Klassenname des UIComponent Objekts.
2. Erzeugen eines Class Objekts des vollklassifizierenden Klassennamens mit Util.load Class((String)Object, Object fallbackClass). Dies führt letztendlich zum Aufruf ClassLoader.loadClass(String) und damit zum gewünschten Class Objekt. Der String Parameter entspricht (String)Object und damit dem componentType aus Object von ApplicationImpl.newThing(Object,Map).
3. Das gewonnene Class Objekt ersetzt den ursprünglichen vollklassifizierenden Klassennanen der ApplicationImpl.componentMap unter demselben Schlüssel componentType als key Variable aus ApplicationImpl.addComponent(componentType, componentClass): Map.put(key, Class).
4. Der Aufruf Class.newInstance() liefert dann das UIComponent Zielobjekt, das zurückgeliefert wird.

Damit ist die gesamte Funktionalität von UIComponentTag.createChild(FacesContext, UIComponentTag, componentId) und auch UIComponentTag.findComponent(Faces Context) abgeschlossen. Mit dem nun zur Verfügung stehenden UIComponent Objekt kann UIComponentTag.doStartTag() mit dem Aufbau bzw. Weiterbau der Baumstruktur der UIComponent Objekte fortfahren. Dies funktioniert ganz einfach mit UIComponentTag.addChild(UIComponent). UIComponentTag ist der Elterntag des gegenwärtigen UIComponent

Tag, UIComponent das gerade erzeugte Tag Repräsentationsobjekt. Z.B. HtmlCommandButton von <h:commandButton>.

Im vorletzten Schritt innerhalb von UIComponentTag.doStartTag() der sich der Encodierung der bisherigen UIComponentTag Baumstruktur widmet, werden alle UIComponentTag.encodeXYZ() Methoden aufgerufen, um den hier generierten HTML Code in den Html ResponseWriter zu schreiben. Am Ende dieses Prozesses, sprich nach der kompletten Encodierung wird der JSF JSP wird der gesamte HTML Code in Form eines <hidden> HTML Elements in den Html ResponseWriter gespeichert. Dieses Element ist der Hauptinhalt des Status Object Objekts für den StateManager.

Eingeleitet wird diese gesamte Prozedur durch UIComponentTag.encodeBegin(). Diese Methode delegiert weiter an UIComponent.encodeBegin() die ihrerseits an den dem konkreten UIComponent Derivat entsprechenden Renderer weiterleitet: Renderer.encodeBegin(FacesContext, UIComponent). Die Implementierung des konreten Renderers, z.B. Button Renderer für die Schaltfläche <h:command>, startet dann unter Verwendung des Html ResponseWriters den eigentlichen HTML Code Genrierungsprozess für dieses JSF Element. Es werden hierfür diverse HtmlResponseWriter.write(...) Methoden verwendet.

Der letzte Schritt von UIComponentTag.doStartTag() ist das Ablegen des gerade erhaltenen UIComponent Objekts auf dem Stack mit UIComponentTag.pushUIComponentTag(). Dieses steht falls Kindelemente vorhanden sind im nächsten Durchlauf von UIComponentTag.doStartTag() als Elterntag zur Verfügung.

Die Callbackmethode ViewTag.doAfterBody() ist nach ViewTag.doStartTag() und vor ViewTag.doEndTag() die dritte Methode innerhalb von loginBody_jsp._jspx_meth_f_view_0(PageContext) mit relevanter Funktionalität. Sie arbeitet eng mit der javax.faces.application.StateManager Implementierungsklasse der Referenzimplementierung zusammen. Sie wird aufgerufen, wenn alle anderen Tags zur ihrer Ausführung gekommen sind, d.h. wenn diese ihrerseits ihre Methoden UIComponentTag.doStartTag() und UIComponent Tag.doEndTag() zur Ausführung gebracht haben, bzw. wenn sie alle in Baumstruktur über den ViewTag traversierbar sind. Die Aufgabe dieser Methode ist es, mit dem StateManager den Inhalt der gesamten JSP in ein Javaobjekt zu schreiben, das Zustandsobjekt, und dieses dann clientseitig zu speichern. Dies geschieht über das weiter oben erwähnte <hidden> HTML Element dessen value Attribut die Baumstruktur als ein langer String unter dem name Attribut mit dem Wert einer Konstante vorhält.

Für diese Aufgabe wird der HtmlReponseWriter und die StateManagerImpl benötigt. Der HtmlReponseWriter wird über den FacesContext bezogen, der StateManagerImpl ist über com.sun.faces.util.Util mit Util.getStateManager(FacesContext) erreichbar. Mit diesen zwei Objekten folgt nun die Einleitung der beiden entscheidenden Schritte:

1. SerializedView = StateManagerImpl.**saveSerializedView**(FacesContext)
2. StateManagerImpl.**writeState**(FacesContext, SerializedView)

In den einführenden Erläuterungen dieses Kapitels wurde die Funktionalität der ersten dieser beiden Methoden bereits angeschnitten. Die entscheidende Aktion in StateManagerImpl.saveSerializedView(FacesContext) ist der Aufruf der Methoden StateManagerImpl.getTreeStructureToSave(FacesContext) und StateManagerImpl.getComponentState ToSave(FacesContext). StateManagerImpl.getTreeStructureToSave(FacesContext) gibt als Resultat ein TreeStructure Objekt zurück, was einfach ein Wrapper um ein UIComponent Objekt ist. Zur Erzeugung dieses TreeStructure Objekts dient eine weitere Hilfsmethode StateManagerImpl.buildTreeStructureToSave(FacesContext,

UIComponent, TreeStructure, Set). TreeStructure ist der Wrapper um UIViewRoot. Der Parameter Set ist durch den Methodenaufruf null.

Das Ziel dieser Methode ist es, ein TreeStructure Objekt zu erstellen, das in sich rekursiv geschachtelt dieselbe Baumstruktur widerspiegelt, wie die des ViewTags, bzw. des UIViewRoots. Dazu bedient sie sich der Rekursion deren Implementierung kurz veranschaulicht wird:

1. Iterator = UIComponent.getChildren().iterator()
2. while(Iterator.hasNext())
3. UIComponent childComp = Iterator.next()
4. TreeStructure childStruct = new TreeStructure(childComp)
5. TreeStructure.addChild(childStruct)
6. StateManagerImpl.buildTreeStructureToSave(FacesContext, childComp, child Struct, HashSet)

StateManagerImpl.getComponentStateToSave(FacesContext) delegiert weiter an UIViewRoot.processSaveState(FacesContext). Diese Methode leistet den Aufbau des Status in Fom des bereits in den einleitenden Ausführungen beschriebenen Object state Objekts. Implementiert wird diese Methode jedoch in UIComponentBase. Die Methode UIComponent Base.processSaveState(FacesContext) ist für das Traversieren aller Kindelemete des gegenwärtigen UIComponent Objekts zuständig:

1. Object[] stateStruct = new Object[2]
2. stateStruct[MY_STATE] = UIComponentBase.saveState(context)
3. Iterator = UIComponentBase.getChildren().iterator()
4. Object[] childState = new Object[Iterator.size()]
5. while(Iterator.hasNext())
6. UIComponent childComp = Iterator.next()
7. stateStruct[CHILD_STATE] = childState
8. childState[i++] = childComp.processSaveState(FacesContext)

Die Hilfsmethode UIComponentBase.saveState(FacesContext) ist für den Aufbau des State Objects zuständig. Hierzu wird ein new Object[8] Array angelegt, der schrittweise gefüllt wird mit Inhalten die den Einführungserläuterungen unter 2.2.1 bei StateManager angegeben wurden.

Damit sind die beiden Methoden innerhalb des Konstruktors SerializedView(getTree StructureToSave(FacesContext), getComponentStateToSave(FacesContext)) für die Objekterzeugung innerhalb der Methode StateManagerImpl.saveSerializedView(Faces Context) abgearbeitet. Das SerializedView Objekt mit dem Status als Object und als TreeStructure Version steht nun in ViewTag.doAfterBody() zur Verfügung. Nun folgt dort der zweite wichtige Schritt: StateManagerImpl.writeState(FacesContext, Serialized View). Auch diese Methode ist eine delegierende Methode nach ResponseStateManager. writeState(FacesContext, SerializedView). Hier werden die beiden Status Objekt Versionen Object und TreeStructure als einfacher Text in ein <hidden> HTML Element überführt.

Verwendet wird hierfür ein ObjectOutputStream und ein ByteArrayOutputStream, der in den ObjectOutputStream übertragen wird. Den Stringinhalt für das value Attribut des <hidden> Elements liefern die Aufrufe ObjectOutputStream.writeObject(Serialized View.getStructure()) und ObjectOutputStream.writeObject(SerializedView.get State()). Der zusammengesetzte <hidden> Elementstring wird zum Abschluss in den Html ResponseWriter des FacesContext geschrieben: HtmlResponseWriter.write(<hidden> Elementstring).

Die Callbackmethode `ViewTag.doEndTag()` markiert das Ende des Encodings einer JSF JSP. Ausser des Aufrufs derselben Methode der Superklasse `UIComponentTag.doEndTag()` gibt es hier keine nennswerte Funktionalität. Wie für jede andere `UIComponentTag` Komponente auch werden zuerst Aufräumarbeiten für das gerade aktuelle Tag Element ausgeführt. Mit `UIComponentTag.popUIComponentTag()` wird zunächst der jeweilige `UIComponentTag` vom Stack des `PageContexts` entfernt. Danach sind die `UIComponents` selbst an der Reihe gelöscht zu werden. Da aufgrund der Rekursion immer die `UIComponentTag.doEndTag()` Methode des tiefsten Elements der `UIComponentTag`, bzw. `UIComponent` Hierarchie zuerst aufgerufen wird, können dessen Kindelemente sofern diese logischerweise ihrerseits keine Kindelemente haben, mit `UIComponent.getChildren().remove(UIComponent child)` gelöscht werden. Dies funktioniert deshalb so, weil z.B. `CommandButtonTag.doEndTag()` vor `ViewTag.doEndTag()` und all dessen übergeordneten Elementen aufgerufen wird, da das `<h:commandButton>` Element weit tiefer geschachtelt ist als `<f:view>`.

Sofern das gegenwärtige `UIComponent` für das Rendern seiner Kindelemente zuständig ist, wird in `UIComponentTag.doEndTag()` nochmals `UIComponentTag.encodeBegin()` und `UIComponentTag.encodeChildren()` aufgerufen. Anschlienßend aber unabhängig davon `UIComponentTag.encodeEnd()`. Diese werden letztlich wieder bis zu den `Renderer`n der entsprechenden `UIComponents` delegiert. Also `Renderer.encodeChildren(FacesContext, UIComponent)` und `Renderer.encodeEnd(FacesContext, UIComponent)`. Die Implementierung dieser Methoden kann je nach `Renderer` nichts oder auch HTML Codegenerierung wie in der Methode `Renderer.encodeBegin(FacesContext, UIComponent)` enthalten.

Nach all diesen komplexen Arbeitsschritten sind nun die Grundlagen vorhanden, den Request Processing Lifecycle in Gang zu setzen.

Schritt 1.2: `Die RestoreViewPhase`

In der ersten Phase werden zwei wichtige Dinge erledigt. Zum ersten, das Beschaffen der `viewId`, also die Webadresse unter der die nächste JSP Seite aufgerufen werden soll. Wie dies funktioniert wurde bereits in den Einführungserläuterungen zur `RestoreViewPhase` behandelt.

Zum zweiten den Aufbau der `UIComponent` Baumstruktur der Zielseite aufgrund der Informationen des Status Objekts, aus dem `<hidden>` Element des JSF JSP Codes. Dies wird durch den Aufruf `ViewHandlerImpl.restoreView(FacesContext, viewId)` eingeleitet. In dieser Methode geht es neben dem Setzen des Character Encodings im Kern darum, an `StateManagerImpl.restoreView(FacesContext, viewId, renderKitId)` zu delegieren. Auch hier werden wieder zwei wichtige Aufgaben erledigt:

1. `UIViewRoot = StateManagerImpl.`**`restoreTreeStructure`**`(FacesContext, viewId, renderKitId)`
2. `StateManagerImpl.`**`restoreComponentState`**`(FacesContext,viewId,renderKitId)`

Das Ziel ist das Beschaffen des Wurzel `UIComponent` und dessen Wrapper `Tree Structure` der JSF Baumelementstruktur der JSP. Um diese zu bekommen wird `ResponseStateManager.getTreeStructureToRestore(FacesContext, viewId)` aufgerufen. Diese Methode besorgt sich den HTML UIComponenthierarchiestring aus dem `<hidden>` HTML Element, über den `FacesContext`, bzw. den `ExternalContext`:

```
String = FacesContext.getExternalContext().getRequestParameterMap().get(
RIConstants.FACES_VIEW)
```

Dieser je nach Inhalt der JSF JSP sehr lange String wird durch `Base64.decode(String.getBytes())` in ein encodiertes `byte` Array tranformiert und in einen `ObjectInputStream`

geladen: `ObjectInputStream = new ObjectInputStream(new ByteArrayInputStream(bytes))`. Aus diesem Stream können dann einfach die beiden Zielobjekte mit `Object = ObjectInputStream.readObject()` und `TreeStructure = ObjectInputStream.readObject()` ausgelesen werden. Das State `Object` wird in der `RequestMap` des `ExternalContext` gespeichert: `RequestMap.put("com.sun.faces.FACES_VIEW_STATE", Object)`. Die `TreeStructure` wird zurückgegeben. Aus dieser kann mit `TreeStructure.createComponent()` das `UIViewRoot` Objekt besorgt werden. Mittels `TreeStructure` kann dann druch `StateManagerImpl.restoreComponentTreeStructure(TreeStructure,UIComponent /*UIViewRoot*/)` die gesamte `UIComponent` Hierarchie aufgebaut werden. Dies funktioniert durch die Rekursion auf die gleiche Art und Weise wie z.B. der geschachtelte Aufbau des Status `Objects`.

1. `Iterator = TreeStructure.getChildren()`
2. `while(Iterator.hasNext)`
3. `TreeStructure child = Iterator.next()`
4. `UIComponent c = child .createComponent()`
5. `UIComponent.getChildren().add(c) // UIComponent aus Methodenparameter`
6. `StateManagerImpl.restoreComponentTreeStructure(child, c) // Rekursion`

Damit ist der Aufbau des kompletten `UIComponent` Hierarchie abgeschlossen und die gesamten Inhalte der JSF JSP stehen damit in Form eines Javaobjekts zur Verfügung.

Schritt 1.3: `Die ApplyRequestValuesPhase`

Mit dem Ende der Abarbeitung der ersten Phase des Request Processing Lifecycle befindet sich das Request Processing in der Methode `LifecycleImpl.execute(FacesContext)` in der Schleife für den Aufruf aller `Phase` Objekte an Stelle zwei. Ausgeführt wird die Phase zwei, sprich `ApplyRequestValuesPhase.execute(FacesContext)` durch den Methodenaufruf `LifecycleImpl.phase((PhaseId)PhaseId.VALUES.get(i),phases[i],FacesContext)`. Der `phases[i]` Parameter beinhaltet mit dem Wert zwei für `[i]` das Phasenimplementierungsobjekt `ApplyRequestValuesPhase`.

Bereits in den Einfühtungserläuterungen angesprochen wurde die einzig wichtige Funktionalität dieser Phase. Den rekursiven Aufruf aller `UIComponent.processDecodes(FacesContext)` aller Kind und Kindeskinder des `UIViewRoot` Objekts. Implementiert wird diese Methode durch `UIComponentBase.processDecodes(FacesContext)`. Diese delegiert nach Ermittlung des für das `UIComponent` passenden `Renderers` weiter an `Renderer.decode(FacesContext, UIComponent)`. Je nach `UIComponent`, bzw. `Renderer`typ ist die `Renderer.decode(FacesContext, UIComponent)` Methode anders implementiert. Für die beiden Textfelder des Beispiels liegt für die `<h:inputText>` Elemente beim `TextRenderer` für das `UIComponent HtmlInputText` die folgende Funktionalität vor: Die Ableitungshierarchie für die Klasse sieht folgendermaßen aus: `TextRenderer extends HtmlBasicInputRenderer extends HtmlBasicRenderer`. Letzterer implementiert letztlich `HtmlBasicRenderer.decode(FacesContext, UIComponent)`. Diese Methode ruft `HtmlBasicInputRenderer.setSubmittedValue(UIComponent, Object)` auf, um dem `HtmlInputText UIComponent` den Wert zu übergeben, den der Benutzer in das `<h:inputText>` Element eingegeben hat. `HtmlBasicInputRenderer.setSubmittedValue(UIComponent, Object)` Setzt einfach mit `HtmlInputText.setSubmittedValue(Object)` das Attribut `HtmlInputText.submittedValue`. Dasselbe passiert natürlich auch mit dem zweiten Texftfeld des Bespiels.

Gesetzt wurde dieser Wert aus dem ursprünglichen value binding Ausdruck `<h:inputText size="30" value="#{loginBean.userName}" />` in der Methode `HtmlBasicRenderer.encode`

End(FacesContext, UIComponent) während der Operationen, die in Schritt 1.1 beschreiben wurden, sprich beim Aufbau der UIComponent Objekthierarchie.

Für das <h:commandButton> Element sieht die ButtonRenderer.decode() Methode hingegen völlig anders aus: Hier steht die Erzeugung eines ActionEvent Objekts und dessen Speicherung in der Event Queue im Vordergrund. In den Einfühtungserläuterungen zum Thema 2.2.4 wurde diese Funktionalität behandelt.

Schritt 1.4: Die ProcessValidationsPhase

Wesentliche Funktionalitäten dieser Phase wurden bereits in Kapitel 2.2.2 besprochen. Daher soll es hier darum gehen, wie es zum Aufruf Validierungsmethoden von Beanklassen kommt, die in der faces-config.xml spezifiziert wurden. Grundlage ist dieselbe Ausgangsposition: Es gibt auf einer JSF JSP ein Element: <inputText id="textfield" validator="#{bean.validatorMethod}">. Durch das Starten der ProcessValidationsPhase mit ProcessValidationsPhase.execute(FacesContext), kommt es zu folgenden Methodenaufrufen:

```
UIViewRoot.processValidators(FacesContext)
 UIComponent.processValidators(FacesContext)
 // HtmlInputText extends UIComponent implements EditableValueHolder
  HtmlInputText.executeValidate(FacesContext)
    HtmlInputText.validate(FacesContext)
      HtmlInputText.validateValue(FacesContext, Object value)
```

Der Aufruf der hierbei in dieser Methode von Interesse ist, lautet MethodBinding.invoke(FacesContext, Object[]). Dieses MethodBinding repräsentiert das den String nach dem validator Attribut des <inputText> Elements. Initilisiert wurde es während der Aktionen, die in Schritt 1.1 beschrieben wurden. D.h. genauer durch die Methode InputTextTag.setProperties(UIComponent). Diese ruft FacesContext.getApplication().createMethodBinding(String, Class[]) auf und delegiert die Erzeugung eines MethodBinding Objekts an ApplicationImpl. Der String Parameter enthält den value binding Ausdruck aus dem validator Attribut des <inputText> Elements. Im Konstruktor von MethodBinding wird dieser Parameter also im Beispiel der Wert #{bean.validatorMethod} umgewandelt zu bean.validatorMethod, sodass per Reflection die entsprechende Methode der Signatur void <valiatorMethodName>(FacesContext, UIComponent, Object) aufgerufen werden kann.

Nach dieser Erzeugung wird das Validator MethodBinding mittels InputTextTag.setValidator(MethodBinding) gesetzt und steht damit später der Methode EditableValueHolder.validateValue(FacesContext, Object) zur Verfügung. Dort kommt MethodBinding.invoke(FacesContext, Object[]) zur Ausführung. So kommt die Validierungslogik des Entwicklers für den Wert des <inputText> Elements zur Anwendung.

Schritt 1.5: Die UpdateUserModelPhase

Der Punkt, der in Kapitel 2.2.2 zur UpdateUserModelPhase noch nicht betrachtet wurde, ist der genaue Vorgang des Aufrufs einer set Methode für ein Attribut einer Beanklasse mittels eines ValueBinding Objekts. Die Ausgangssituation ist wie bei der ProcessValidationsPhase die gleiche. Es gibt auf einer JSP ein UI Element dessen UIComponent Repräsentationsobjekt das EditableValueHolder Interface implementiert, wie das HtmlInputText Element:

```
<inputText id="textfield" value="#{bean.attribute}">
```

Durch die Ausführung von UpdateUserModelPhase.execeute(FacesContext) entsteht der folgende Aufrufshierarchie für jede EditableValueHolder UI Komponente bis hin zur

eigentlichen Updatemethode für das Beanattribut, das durch das `<inputText>` Element referenziert wird.

```
UIViewRoot.processUpdates(FacesContext)
 UIComponent.processUpdates(FacesContext)
 // HtmlInputText extends UIComponent implements EditableValueHolder
  HtmlInputText.updateModel(FacesContext)
```

Über die Basisklasse `UIComponentBase` verfügt `HtmlInputText` durch die Methode `UIComponentBase.getValueBinding(String)` über alle `ValueBindings`, die ihr bei der Erzeugung der `UIComponent` Elementhierarchie in der `HtmlInputText.setProperties(UIComponent)` Methode durch `ValueBinding = Util.getValueBinding(String wert)` und `HtmlInputText.setValueBinding("value", ValueBinding)` hinzugefügt wurden. Auf genau dieses `ValueBinding` wird in `HtmlInputText.updateModel(FacesContext)` zugegriffen: `ValueBinding = UIComponentBase.getValueBinding(`**`"value"`**`)`. Gesetzt wird der neue Wert, den der Benutzer in das Textfeld eingetragen hat mit der nächsten Anweisung `ValueBinding.setValue(FacesContext, String)` wobei der `String` Parameter diesen Wert enthält.

Schritt 1.6 `Die InvokeApplicationsPhase`

Die ist der Teil des Request Processing Lifecycle, in dem die Ergeinisse aller Phasen an die Auslöser `UIComponent` Objekte gefeuert werden. Angestoßen wird dies in der gewohnten Schlichtheit durch die folgende Aufrufkette:

```
InvokeApplicationPhase.execute(FacesContext)
  UIViewRoot.processApplication(FacesContext)
    UIViewRoot.broadcastEvents(FacesContext, PhaseId.INVOKE_APPLICATION)
      while(eventListNotEmpty)
        FacesEvent = eventList.get(i)
        UIComponent = FacesEvent.getComponent()
        UIComponent.broadcast(FacesEvent)
```

Die Internas, die von dem Aufruf der letzten Methode dieser Kette ablaufen, wurden im Abschnitt 2.2.4 behandelt.

Schritt 2: `Lifecycle.render(FacesContext)`

Schritt zwei behandelt die letzte Phase des Request Processing Lifecycle, die `Render ResponsePhase`. Ausgelöst wird sie in `FacesServlet.service(ServletRequest, Servlet Response)` durch `LifecycleImpl.render(FacesContext)` direkt nach `LifecycleImpl.execute(FacesContext)`. Im Grunde macht diese Methode nichts anderes als die zuletzt genannte Methode, da sie ebenfalls einfach `LifecycleImpl.phase(PhaseId, Phase, Faces Context)` ausführt, genauer `LifecycleImpl.phase(PhaseId.`**`RENDER_RESPONSE`**`, Phase, FacesContext)`. Durch `RenderResponsePhase.execute(FacesContext)` wird hier die letzte Phase des Request Processing Lifecycles gestartet, die nur einen wesentlichen Aufruf hat:

```
FacesContext.getApplication().
            getViewHandler().renderView(FacesContext,
                                        FacesContext.getViewRoot())
```

Es handelt sich also um eine Delegation an `ViewHandler().renderView(FacesContext, FacesContext.getViewRoot())`. Dort wiederum gibt es für das Verständnis nur zwei wichtige Aktionen: `UIViewRoot.setViewId(String)` und `ExternalContext.dispatch(String)`.

Der `String` Parameter enthält die `viewId` für die nächste JSP, also einen Request URI wie z.B. `http://localhost:8080/view/newPage.jsp`.

Um auf die nächste Seite zu verzweigen, wird wie in Struts auch bei dieser Aufgabe der `javax.servlet.RequestDispatcher` benutzt. Die nächste Seite kommt dann durch `Request Dispatcher.forward(ServletRequest, ServletResponse)` zur Anzeige. Damit ist das Ende des Request Processing Lifecycle erreicht. Sobald der Benutzer auf einen Button oder einen Link klickt, beginnt alles was in Kapitel 2.4 besprochen wurde, wieder von vorne.

2.5 Zusammenfassung

Die Java Server Faces von Sun dienen wie Struts dem Zweck einer standardisierten Webapplikationsentwicklung. Auch dieses Framwork basiert auf dem MVC Muster mit einem kleinen Unterschied zu Struts: Die Integration von Controllern in die eine Modelklasse in Form von Methoden, die auch das Verzweigen von Viewkomponenten übernehmen.

Ebenfalls gleich sind die zwei großen Säulen, das Hochfahren der Anwendung und die Requestbearbeitung. Während das Hochfahren vom Prinzip her identisch mit dme von Struts ist, hier das Auslesen der fast namensgleichen `faces-config.xml` und die Übertragung der dort hinterlegten Informationen zum in ein Javaobjekt `FacesConfigBean`, gibt es bei der Reqlisierung des Request Processing einen gänzlich anderen Ansatz.

Das Herzstück des Requestbearbeitung in JSF ist der Request Processing Lifecycle. Er besteht aus sechs voneinander unabhängigen Phasen, die je nach Komplexität des Requests alle durchlaufen werden, oder im Falle eines einfachen Reloads der Seite abgesehen von der ersten und letzten Phase ausgelassen werden können.

Ein besonderes Feature der Java Server Faces ist deren Trennung von abstrakten Schnittstellen auf der einen und der Implementierung auf der anderen Seite. Dies ermöglicht es, dass mehrere Hersteller JSF Implementierungen zur Verfügung stellen können, die im Idealfall beliebig austauschbar sind. Eine weitere Stärke ist der Fokus von JSF auf GUI Elemente. Das Framework bietet mit seinem GUI Komponentenmodell die Möglichkeit, GUI Komponenten zu entwickeln, die von dem HTML Standard nicht berücksichtigt werden, wie z.B. Trees oder Registerkarten. Auch auf diesem Gebiet gibt es bereits verschiedene Anbieter, die Weboberflächen bereichern.

3 Komponenten und Strukturen des SAP CRM

Nach der Untersuchung der Open Source Webentwicklungsframeworks Struts und JSF, ist das Thema der Kapitel drei und vier, die Softwareentwicklung mit Weboberflächen unter SAP zu beleuchten. Dabei gibt es immer zwei große Bereiche. Die eine Seite ist die Entwicklung auf dem SAP System, sprich das Programmieren von ABAP Funktionsbausteinen, das Customizing und falls nötig auch das Erstellen eigener SAP Datenbanktabellen. Um diese Aufgaben erledigen zu können, sind detaillierte Kenntnisse über Geschäftsobjekte erforderlich, die es über einen großen Bereich, wie z.B. das Customer Relationship Management gibt. Die vier zentralen Geschäftsobjekte sind beim CRM der Geschäftspartner, der Geschäftsvorgang, das Produkt und die Konditionstechnik. Eingegangen wird in dieser Arbeit auf die beiden ersten.

Die zweite große Säule bei der Entwicklung von SAP Webapplikationen ist die Nutzung von ABAP Funktionsbausteinen für den Zweck der Anwendung, womit sich Kapitel vier befasst. Um das Kapitel vier zu verstehen ist es nicht unbedingt erforderlich, dieses Kapitel zu lesen, da die Details zum Geschäftspartner und Geschäftsvorgang zumindest keine entscheidende Rolle spielen. Wenn der Leser jedoch eine umfassende Übersicht über diese zwei Entitäten und CRM von der Datenseite her, weniger von der betriebswirtschaftlichen Seite, bekommen möchte, wird er hier Informationen und Einblicke erhalten.

3.1 Kapitelübersicht

Die Literatur von SAP zum Thema SAP CRM 4.0 wie z.B. [7], beschäftigt sich vornehmlich mit dem betriebswirtschaftlichen Nutzen und der Anwendung im täglichen Geschäft. Dies beinhaltet eine CRM Einführung aus SAP Sicht, die Umsetzung dieses Wissens in SAP CRM Geschäftsprozesse und schlussendlich die Intagration in die SAP Netwaever Integrations-architektur.

Um neben der bereits vorhandenen Literatur weniger angesprochene Erkenntnisse zum SAP CRM zu gewinnen, liegt der Schwerpunkt dieses Kapitels auf der reinen Datenseite. Als Informationsbasis dienten für dieses Kapitel ein SAP CRM 4.0 System und [8] und [9].

Die wichtigsten Elemente, mit denen man beim CRM immer in Berührung kommt, sind Geschäftspartner und Geschäftsvorgänge, die die geschäftliche Interaktion zwischen ihnen und den Kunden beinhalten. Daten über Geschäftspartner und Geschäftsvorgänge müssen logisch und übersichtlich gespeichert und gepflegt werden, um sie anschließend in strukturierter Form betriebswirtschaftlich nutzbar zu machen, d.h. sie in Information umwandeln zu können. Die

Information ist das verwertbare Endprodukt, das die Nutzer für ihre unterschiedlichen Tätigkeiten nutzen können.

Die strukturierte Form der Daten, also die Grundlage für alles Weitere ist der Kern des Kapitels drei. Um diese Strukturierung zu ergründen wird detailliert der Aufbau der Hauptentitäten Geschäftspartner und Geschäftsvorgang untersucht. Dies umfasst an deren Aufbau beteiligte Tabellen und deren Verbindungen untereinander. Es ist nicht das Ziel dieses Abschnitts detailliert jedes Attribut einer Tabelle zu erörtern, sondern es wird ein umfassender Überblick über die Breite gegeben werden, mit wesentlichen Informationen über die wichtigsten Inhalte der Tabellen neben den Tabellenbeziehungsattributen.

Auf diese Weise gewinnt der Leser einen übersichtsschaffenden Eindruck über die naturgemäße Komplexität von Geschäftspartnerbeziehungen und Geschäftsvorgängen mit all deren möglichen Facetten, die in einem relationalen Datenmodell abgebildet werden müssen. Im Text werden Tabellen und deren Attribute mit der Schrift Century Gothic der Größe zehn vom restlichen Text kenntlich gemacht. Die Notation Tabelle::Tabellenfeld bedeutet, dass die Tabelle das Tabellenfeld hinter den zwei Doppelpunkten besitzt.

3.2 Datenstrukturen und Beziehungen von SAP CRM Komponenten

Bevor auf die Geschäftsobjekte eingegangen wird, müssen Grundlagen und Grundbegriffe besprochen werden, die beim SAP CRM verwendet werden. Es gibt erstens verschiedene Objekttypen und Sichten eines Geschäftsobjekts. Zweitens werden SAP Datenbanktabellen in Komponente, Subtyp, Extension und Set unterschieden. Ferner gibt es bei Tabellen verschiedene Generalisierungstypen.

3.2.1 Aufbau und Struktur von SAP Datenbanktabellen

Der Inhalt dieses Abschnitts sind zunächst einführende und grundlegende Informationen zu SAP CRM Tabellen. Dabei geht es um grundsätzliche Strukturierungsprinzipien, die grundsätzlich für alle SAP CRM Tabellen gültig sind. Mit diesem Hintergrund werden die Tabellenstrukturen- und Beziehungen analysiert, die für das Verständnis des Aufbaus und der Verbindungen von Geschäftspartner- und Geschäftsvorgangsdatenbanktabellen wichtig sind.

Die Strukturierung des SAP CRM

Die folgenden Ausführungen beziehen sich auf das SAP CRM Version 4.0. Die Strukturierung lässt sich durch ein zwei dimensionales karthesisches Koordinatensystem mit einer x- und y-Achse veranschaulichen.

Auf der y-Achse werden die betriebswirtschaftlichen Sachgebiete abgetragen. Diese werden auch als Objekte bezeichnet. Sie dürfen nicht mit den CRM Objekten verwechselt werden. Dazu gehören:

- Geschäftspartner
- Geschäftsvorgang

Die x-Achse wird in vier Spalten aufgeteilt. Jede Spalte steht für eine betriebswirtschaftliche Qualität und wird auch als Komponente des SAP CRM bezeichnet. Die betriebswirtschaftliche Qualität steht für die individuelle Sicht des Anwenders, bzw. für den speziellen Bezug der

Betrachtung auf das betriebswirtschaftliche Sachgebiet. Jede Komponente besteht aus CRM-Objekten, die durch Datenbanktabellen abgebildet werden.

Für jedes Objekt im SAP CRM gibt es die folgenden vier Komponenten, bzw. Sichten.

Komponente/Sicht	Beschreibung
Customizing Objekte → **zeitraumbezogen**	Customizing Objekte charakterisieren die individuellen, spezifischen Anforderungen der SAP CRM Installation, bzw. Applikation. Für jede Branche und jedes Unternehemen müssen diese Customizing Tabellen im Rahmen der SAP CRM Einführung individuell designed werden. D.h. welche Felder und welche Datentypen die einzelnen Tabellen enthalten müssen. Damit regeln die Customizing Objekte den spezifischen betriebswirtschaftlichen Ablauf des Unternehmens. Darüber hinaus dienen Customizing Objekte auch für Einstellungen und Konfiguration
Strategische Objekte → **zeitraumbezogen**	Strategische Objekte beinhalten Informationen über • organisatorische Einheiten wie z.B. Bundesländer, Regionen, Unternehmen, Abteilungen, Geschäftspartner, … • (im)materielle Stammobjekte wie z.B. Angebot, Auftrag, Produkt, Dienstleistungen, … • Geschäftstätigkeitsregeln, wie z.B. Bonität prüfen, Zugriffsrechte auf Informationen, Geschäftsprozesssteuerung, Ordnungsvorschriften, u.a.m.
Administrative Objekte → **zeitraumbezogen**	Administrative Objekte bilden Mengen– und Wertvorschriften, sowie –Bedingungen ab. Z.B. Informations- und Preisfindung, Rabattfindung, u.a.m.
Operative Objekte → **zeitpunktbezogen**	Operative Objekte umfassen Informationen über Dokumente und Geschäftsvorfälle. Z.B. Fakturierung, Verkauf, Versand

Tabelle 5: Die vier Sichten auf die Geschäftsobjekte des SAP CRM 4.0. Diese bilden die grundlegende Strukturierung der Hauptentitäten im SAP CRM.

Die folgende Tabelle liefert eine Übersicht, zu welchen betriebswirtschaftlichen Objekten welche Sichten, bzw. Komponenten mittels CRM Objekten, also Datenbanktabellen vorhanden sind.

betriebswirtschaftl. Sachgebiet				
Geschäftspartner	vorhanden	vorhanden		
Geschäftsvorgang	vorhanden			vorhanden
Konditionstechnik	Vorhanden	vorhanden	vorhanden	vorhanden

Produkt	Vorhanden	vorhanden			Komponente/ betriebswirtschaftl. Qualität
	Customizing	**Strategisch**	**Administrativ**	**Operativ**	

Abbildung 11: Einordnung von betriebswirtschaftlichen Sachgebieten in die vier Geschäftsobjekte SAP CRM Architektur

3.2.2 Häufig verwendete Fachbegriffe

Bei den folgenden Begriffen handelt es sich um Strukturierungsweisen, die den Aufbau der CRM Entitäten charakterisieren und deren relationale Abbildung in Tabellen regeln. Für alle betriebswirtschaftlichen Gesichtspunkte, die in CRM Objekten, bzw. Entitäten stecken, müssen sehr viele Tabellen miteinander in konsistenten Beziehungen stehen. Diese gesamte CRM Objekt Tabellenstruktur muss so flexibel und abstrakt gültig sein, dass sie branchenspezifische Besonderheiten abdeckt und dabei kein individuelles Customizing erforderlich macht. Zu diesem Zweck gibt es Komponenten, Subtypen, Extensions und Sets.

Strukturierungstyp	Beschreibung
Komponente	Eine Sammlung von betriebswirtschaftlichen, zusammengehörigen Eigenschaften, die eine CRM Entität charakterisieren. Eine CRM Entität lässt sich als ein Baukasten oder einen Ordnungsrahmen beschreiben, der aus beliebig vielen und komplexen Bausteinen besteht. So ist z.B. bei einer Entität Geschäftspartner eine der vielen Komponenten die Geschäftspartnerbankverbindung. Ist eine Komponente noch von anderen Entitäten, oder wiederum von anderen Komponenten nutzbar, also nicht entitätenspezifisch, bezeichnet man sie als **übergreifende** Komponente. Komponenten lassen sich generell nach den drei folgenden Typen unterscheiden
Subtyp	Der Subtyp entspricht einer Generalisierung zweier Daten-banktabellen. Der Subtyp hat alle Eigenschaften seines Basistyps, und definiert neue nur für diesen Untertyp geltende, zusätzliche Eigen-schaften. Beim Geschäftspartner beispielsweise gibt es die Subtypen NATÜRLICHE PERSON, ORGANISATION und GESCHÄFTSPARTNERGRUPPE.
Extension	Für spezifische betriebswirtschaftliche Sachverhalte reichen die Basisinformationen der Entitätshaupttabellen nicht aus. Zu diesem Zweck gibt es die Extensions(- Tabellen). Diese Tabellen erweitern die Eigenschaften einer Entität mit zusätzlichen Attributen um den gewünschten kontextbezogenen Sachverhalt abzudecken. Zwischen Entität und Extension besteht immer eine 1:1 Beziehung. Die Geschäftspartnerbankverbindung ist ein Beispiel für eine Extension für eine Entität Geschäftspartner. Ein Datensatz aus dieser Extensiontabelle gehört atomar zu einem bestimmten Geschäftspartner.
Set	Ein Set ist eine Sammlung von semantisch zusammengehörigen Tabellenattributen, die in mindestens einer Tabelleninstanz, also einem Datensatz identisch vorkommen. Aus den Gründen der Redundanzvermeidung und der Wiederverdungsmöglichkeit werden diese Eigenschaften in eine weitere Tabelle ausgelagert und tauchen somit nicht

	mehr in den Ursprungstabellen auf. Zwischen Tabelle und Set(- Tabelle) besteht naturgemäß eine n:m Beziehung. Um diese Relation ordnungsgemäß aufzulösen besteht zwischen Set und Tabelle eine Zuordnungstabelle. Die Lebensdauer eines Sets endet, wenn keine Tabelleinstanz mehr auf eine Setinstanz verweist.

Tabelle 6: Tabellentypen SAP CRM. Jede Tabelle eines SAP CRM Geschäftsobjekts gehört einem dieser Typen an.

Generalisierungstypen

Es gibt vier Typen der Generalisierung als Beziehung zwischen SAP Datenbanktabellen. Unterschieden werden sie nach den beiden Kriterien der Voll- oder Unvollständigkeit und nach disjunkter oder nicht disjunkter Generalisierung. Die schwarzen Punkte stellen die Entitäten, also die Datenbanktabellen dar, die gestrichelten Kreise bestimmen einen Subtyp von Entitäten.

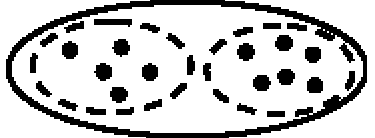
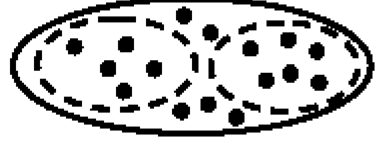
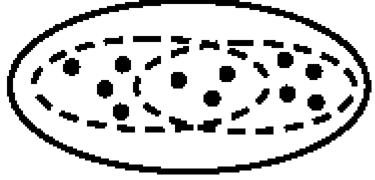
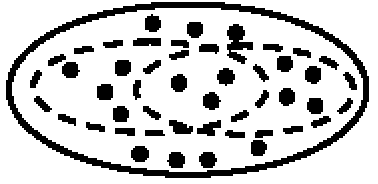
vollständige disjunkte Spezialisierung	**nicht vollständige disjunkte Spezialisierung**
Vollständige, disjunkte Spezialisierung bedeutet, dass jede Entität genau von **einem** Subtyp ist. Dies entspricht einer Einfach-Vererbungshierarchie einer Programmiersprache, wie sie z.B. in Java vorkommt. Jede Klasse erbt mindestens, bzw. auf jeden Fall von der Klasse Object. 	Vollständige, nicht disjunkte Spezialisierung bedeutet, dass eine Entität von **einem oder keinem** Subtypen erben muss. 
vollständige nicht disjunkte Spezialisierung	**nicht vollständige nicht disjunkte Spezialisierung**
Vollständige, nicht disjunkte Spezialisierung bedeutet, dass eine Entität von **einem oder mehreren** Subtypen erben muss. 	Nicht vollständige, nicht disjunkte Spezialisierung bedeutet, dass eine Entität von **keinem oder mehreren** Subtypen erben muss. Dies entspricht einem Sachverhalt wie in der Programmiersprache C++. 

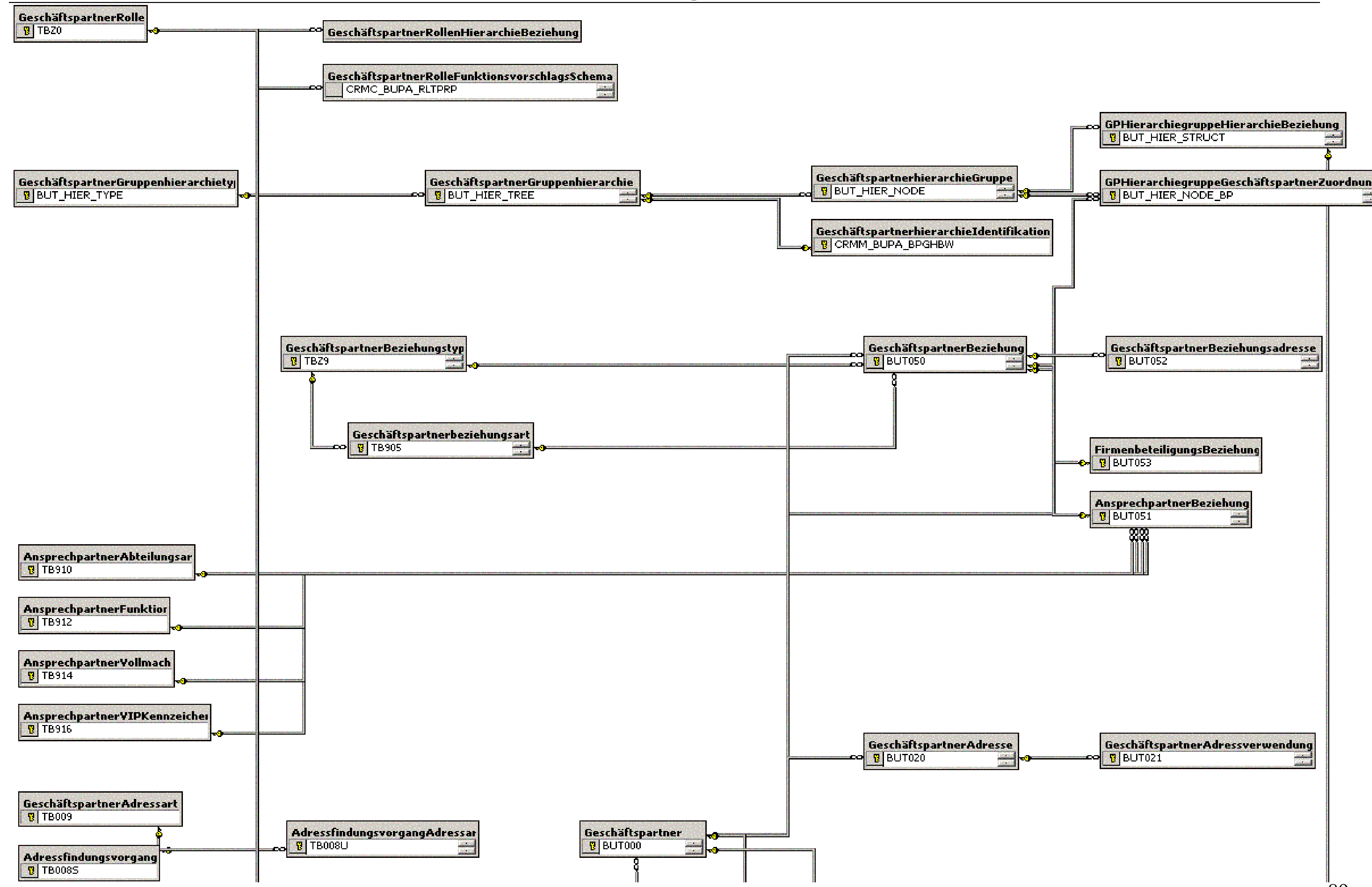
Tabelle 7: Generalisierungstypen von SAP CRM

3.3 Der Geschäftspartner

Das Schaubild der beiden folgenden Seiten bildet die Struktur der Tabellen des Geschäftspartners ab. Um den Ausführungen dieses Kapitels besser folgen zu können, wird empfohlen ein SAP CRM 4.0 System per SAP GUI zu nutzen.

Das Zentrum der Tabellenstruktur des Geschäftspartners bildet die Tabelle BUT000. Innerhalb von SAP CRM 4.0 gibt es drei Ausprägungen eines Geschäftspartners: NATÜRLICHE PERSON, ORGANISATION und die GESCHÄFTSPARTNERGRUPPE.

Es liegt bei diesen Ausprägungen eine vollständige und disjunkte Generalisierung vor. Abgebildet wird diese kleine Vererbungshierarchie durch das `TYPE` Attribut von `BUT000`. Es ist also keine separate Tabelle für jeden Subtyp vorhanden. Das `TYPE` Attribut definiert einen Geschäftspartner als NATÜRLICHE PERSON mit dem Wert 0. Einer ORGANISATION ist dessen `TYPE` Attribut der Schlüsselwert 1 zugeordnet, der GESCHÄFTSPARTNERGRUPPE der Wert 2.


GeschäftspartnerRolle
TBZ0
GeschäftspartnerRollenHierarchieBeziehung
GeschäftspartnerRolleFunktionsvorschlagsSchema
CRMC_BUPA_RLTPRP
GPHierarchiegruppeHierarchieBeziehung
BUT_HIER_STRUCT
GeschäftspartnerGruppenhierarchiety
BUT_HIER_TYPE
GeschäftspartnerGruppenhierarchie
BUT_HIER_TREE
GeschäftspartnerhierarchieGruppe
BUT_HIER_NODE
GPHierarchiegruppeGeschäftspartnerZuordnung
BUT_HIER_NODE_BP
GeschäftspartnerhierarchieIdentifikation
CRMM_BUPA_BPGHBW
GeschäftspartnerBeziehungstyp
TBZ9
GeschäftspartnerBeziehung
BUT050
GeschäftspartnerBeziehungsadresse
BUT052
Geschäftspartnerbeziehungsart
TB905
FirmenbeteiligungsBeziehung
BUT053
AnsprechpartnerBeziehung
BUT051
AnsprechpartnerAbteilungsar
TB910
AnsprechpartnerFunktior
TB912
AnsprechpartnerVollmach
TB914
AnsprechpartnerVIPKennzeicher
TB916
GeschäftspartnerAdresse
BUT020
GeschäftspartnerAdressverwendung
BUT021
GeschäftspartnerAdressart
TB009
Adressfindungsvorgang
TB008S
AdressfindungsvorgangAdressar
TB008U
Geschäftspartner
BUT000

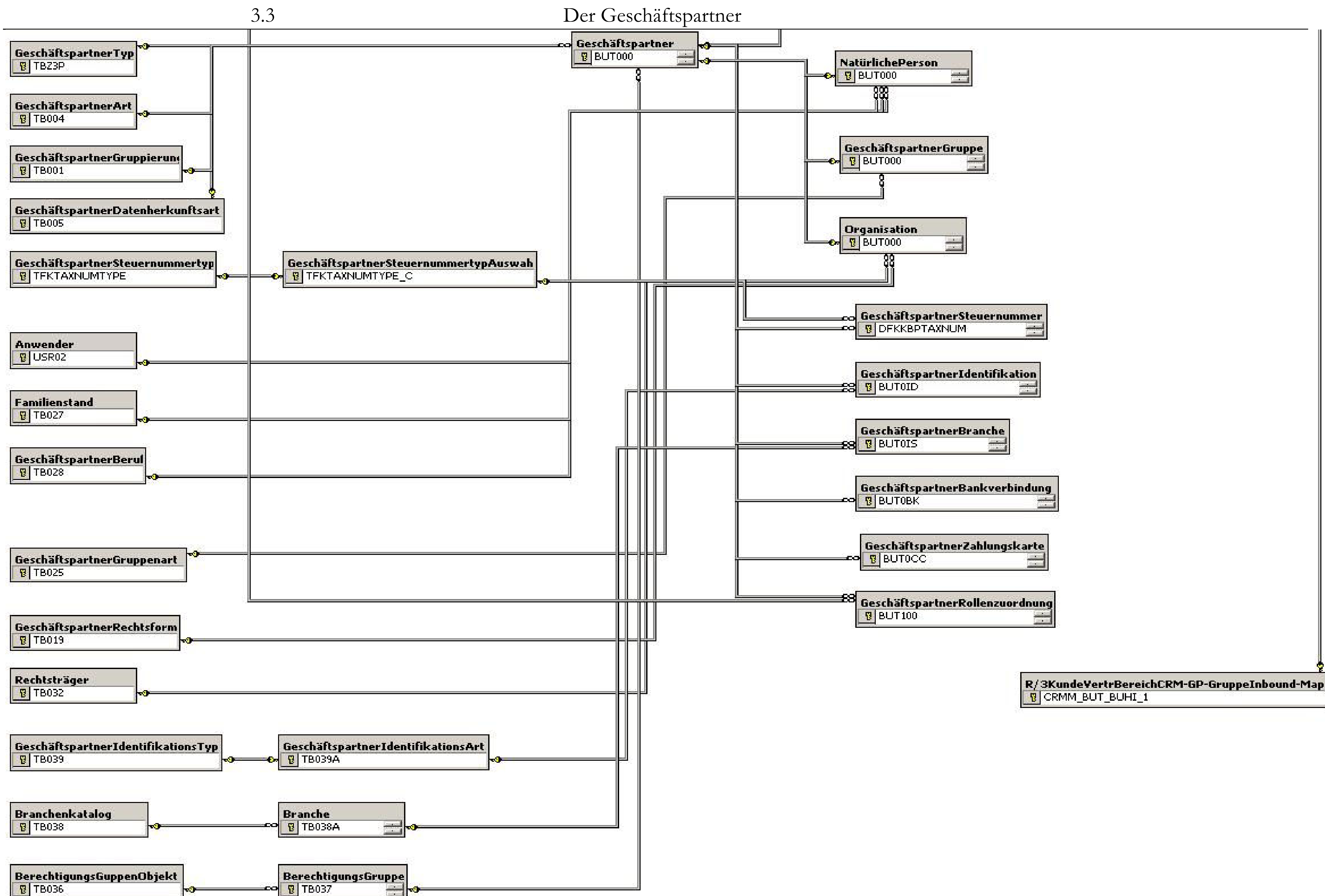


Abbildung 14: Das Entity Relationship Modell des SAP CRM Geschäftpartners.

3.3.1 Umgebungstabellen des Geschäftspartners

Für das Verständnis von Inhalt und Bedeutung der Extension Tabellen wird besonderes Augenmerk auf die Realisierung der Verknüpfungen zur BUT000 Tabelle gelegt. Hierzu werden die betreibswirtschaftliche Bedeutung und die Eigenschaften jeder Kategorie und ihrer beteiligten Tabellen dargelegt. Anschließend folgt die Analyse der technischen Realisierung, d.h. die Erläuterung der entscheidenen Felder der Datenbanktabellen und deren Beziehungen untereinander und zur Haupttabelle BUT000. SAP CRM definiert für den Geschäftspartner die folgenden Extensions:

Kategorie	Tabellenname	Tabellenbezeichnung
GESCHÄFTSPARTNERGRUPPEN-HIERARCHIE	BUT_HIER_TREE	Geschäftspartner Gruppenhierarchie
	BUT_HIER_NODE	Geschäftspartner Hierarchiegruppe
	CRMM_BUPA_BPGHB	Geschäftspartner Gruppenhierarchieidentifikation
	BUT_HIER_STRUCT	Geschäftspartner Gruppenhierarchiebeziehung
	BUT_HIER_NODE_BP	Geschäftspartner Hierarchiegruppe GP. Zuordnung
GESCHÄFTSPARTNERBEZIEHUNG	BUT050	Geschäftspartner Beziehung
	BUT052	Geschäftspartner Beziehungsadresse
	BUT053	Firmenbeteiligungsbeziehung
	BUT051	Ansrechpartnerbeziehung
GESCHÄFTSPARTNER	BUT020	Geschäftspartner Adresse
	BUT021	Geschäftspartner Adressverwendung
	DFKKBPTAXNUM	Geschäftspartner Steuernummer
	BUT0ID	Geschäftspartner Identifikation
	BUT0IS	Geschäftspartner Branche
	BUT0BK	Geschäftspartner Bankverbindung
	BUT0CC	Geschäftspartner Zahlungskarte
	BUT100	Geschäftspartner Rollenzuordnung
R/3 Kunde CRM Geschäftspartnergruppe Mapping	CRMM_BUT_BUHI_1	R/3 Kunde Vertriebsbreich CRM GP. Gruppe Inbound Map

Tabelle 8: SAP CRM Extension Tabellen des Geschäftspartners

Die folgenden Abbildungen, die Tabellenbeziehungen darstellen, verwenden nur die Tabellennamen im CRM System. Um nachzusehen, um welche Tabelle es sich handelt, ist in der vorherigen Tabelle die zugehörige Tabellenbezeichnung ersichtlich.

Jeder Geschäftspartner wird einer Hierarchiegruppe zugeordnet. Eine Hierarchiegruppe dient zur Ordnung und Strukturierung von verschiedenen Arten von Geschäftspartnern und deren Bedeutung für den Umsatz eines Unternehmens. So unterscheiden sich z.B. Privatpersonen aus dem Privatkundengeschäft und Großkunden wie Firmen, fundamental hinsichtlich ihrer Kaufkraft –und Menge, der Kaufhäufigkeit und dem rechtlichem Status. Daher ist es sinnvoll, jeden Kunden in einer Hierarchie einzuordnen, die alle individuellen und definierbaren Unterscheidungskriterien berücksichtigt. Es ist möglich, einen Geschäftspartner mehreren Hierarchiegruppen zuzuordnen.

Die Geschäftspartnerhierarchiegruppen können ihrerseits in Geschäftspartnergruppenhierarchien strukturiert und hierarchisch angeordnet werden. Diese Strukturierung und Anordnung

wird mittels sogenannter Identifikationen ermöglicht, die jeder Geschäftspartnerhierarchiegruppe automatisch zugewiesen wird. Klassifiziert werden Geschäftspartnergruppenhierarchien durch Geschäftspartnergruppenhierarchietypen, die festlegen für welche Zwecke sie verwendet werden.

Mit dem Datenbanktabellenfeld BUT_HIER_NODE_BP::BU_PARTNER_GUID der Tabelle BUT_HIER_NODE_BP ist die Haupttabelle BUT000 über eine Hierarchiegruppenzugeordnung einer Geschäftspartnerhierarchiegruppe zugeordnet. Diese Verknüpfung ist folgendermaßen realisiert:

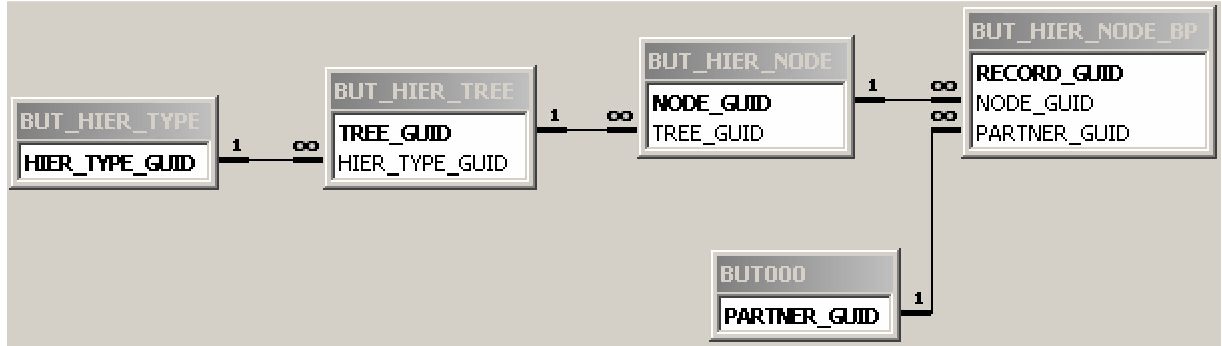


Abbildung 12: Zuordnung einer Hierarchiegruppe zu einem Geschäftspartner

Jeder Geschäftspartner, also jeder Datensatz in der Tabelle BUT000, ist per 1:n Fremdschlüsselreferenzierung mit dem Attribut BUT000::PARTNER_GUID in der Tabelle BUT_HIER_NODE_BP mit deren Feld BUT_HIER_NODE_BP::PARTNER_GUID vorhanden. Diese Tabelle ist eine n:m Beziehungsauflösungstabelle zwischen BUT000 und BUT_HIER_NODE. Ebenfalls über eine 1:n Fremdschlüsselbeziehung sind BUT_HIER_NODE_BP und BUT_HIER_NODE über deren Attribut NODE_GUID verknüpft. Der Geschäftspartner Hierarchiegruppen GP. Zuodnungstabelle BUT_HIER_NODE_BP muss eine Hierarchiegruppe zugewiesen werden, die als Hilfstabelle dem Geschäftspartner eine Hierarchiegruppe zuordnet.

Nach dem gleichen Muster werden Hierarchiegruppen in Geschäftspartnergruppenhierarchien eingeordnet. Die Tabelle BUT_HIER_NODE besitzt als Fremdschlüssel das Feld TREE_GUID. Dieses Attribut ist der Primärschlüssel der BUT_HIER_TREE Tabelle. Somit wird jeder Hierarchiegruppe eine Gruppenhierarchie zugeordnet.

Der Bezug zum Customizing wird mit der 1:n Fremdschlüsselbeziehung zwischen BUT_HIER_TREE::HIER_TYPE_GUID und BUT_HIER_TYPE::HIER_TYPE_GUID hergestellt. Die Tabelle BUT_HIER_TREE ist im Gegensatz zu den anderen Tabellen aus Abbildung 12 eine Customizing Tabelle, in der Geschäftspartnergruppenhierarchietypen definiert werden müssen, die dem jeweiligen Unterneh-men angepasst sind.

Zwischen Geschäftspartnern gibt es Geschäftspartner**beziehungen**. Diesen Beziehungen können jeweils eine Bedeutung zugeordnet werden. Beziehungsbedeutungen werden mit Beziehungstypen beschrieben und lauten beispielsweise: „Geschäftspartner A ist Mitarbeiter von Organisation B“, oder „Vertriebsleiter B ist Ansprechpartner für Einkaufsleiter C“.

Bestimmte Geschäftspartnerbeziehungen können eine Geschäftspartnerbeziehungsadresse haben. Z.B. Person A ist Rechnungsempfänger von Person B. Hier muss B wissen, wohin er sich wenden muss, um mit Person A in Kontakt treten zu können. Hat die Geschäftspartnerbeziehung den Geschäftspartnerbeziehungstyp „ist Vorgesetzter von“, so kann es hier keine Geschäftspartnerbeziehungsadresse geben, da dies aus wirtschaftlicher Sicht überflüssig ist.

Es gibt zwei verschiedene, erweiternde Typen von Geschäftspartnerbeziehungen. Dies sind die Firmenbeteiligungsbeziehung und die Ansprechpartnerbeziehung, für die jeweils eine separate Tabelle im SAP CRM 4.0 implementiert ist.

Die Ansprechpartnerbeziehung erweitert die Geschäftspartnerbeziehung um Informationen wie die Funktion des Ansprechpartners, den Umfang der Autorisierung in Rechtsfragen für die Firma, oder in welcher Firmenabteilung der Ansprechpartner tätig ist. Bei der Firmenbeteiligungsbeziehung stehen sich immer eine Organisation und ein Anteilseigner, bzw. eine Gruppe von Anteilseignern gegenüber. Somit bildet diese Art einer Geschäftspartnerbeziehung immer eine Firmenbeteiligung ab.

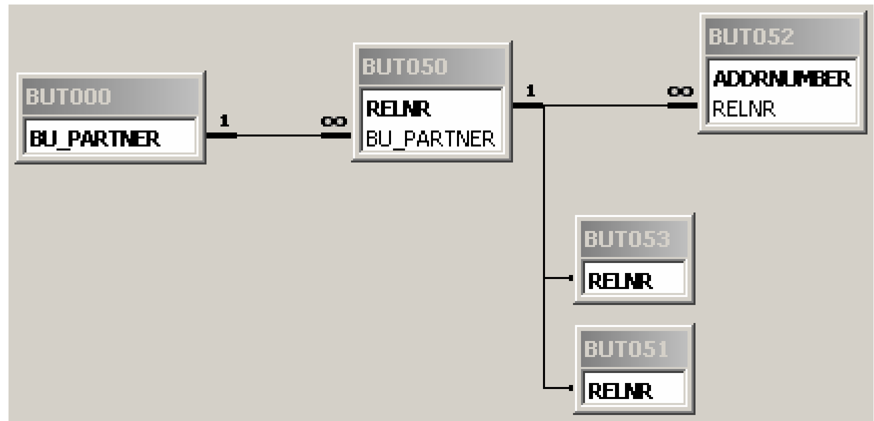


Abbildung 13: Zuordnung einer Geschäftspartnerbeziehung zu einem Geschäftspartner

Jede Geschäftspartnerbeziehung ist mit ihrem Fremdschlüsselattribut BUT050::BU_PARTNER mit der Geschäftspartnertabelle BUT000 verbunden. So sind jedem Geschäftspartner Informationen über seine Geschäftspartnerbeziehung zugeordnet, die für den Kontakt zu anderen Geschäftspartnern von Bedeutung sind. Ist eine Geschäftspartnerbeziehung vom Typ Ansprechpartnerbeziehung, gibt es Attributfelder, die sich in die folgenden Kategorien einteilen lassen:

- Allgemeine Daten über den Ansprechpartner wie
  - Abteilung
  - Funktion und Beschreibung
  - Vollmacht
  - Bemerkungen zu einem Partner
- Adressdaten
  - Telefonnummer: Vorwahl, Anschluss, Nebenstelle
  - Faxnummer: Vorwahl, Anschluss, Durchwahl
  - E-Mail Adresse

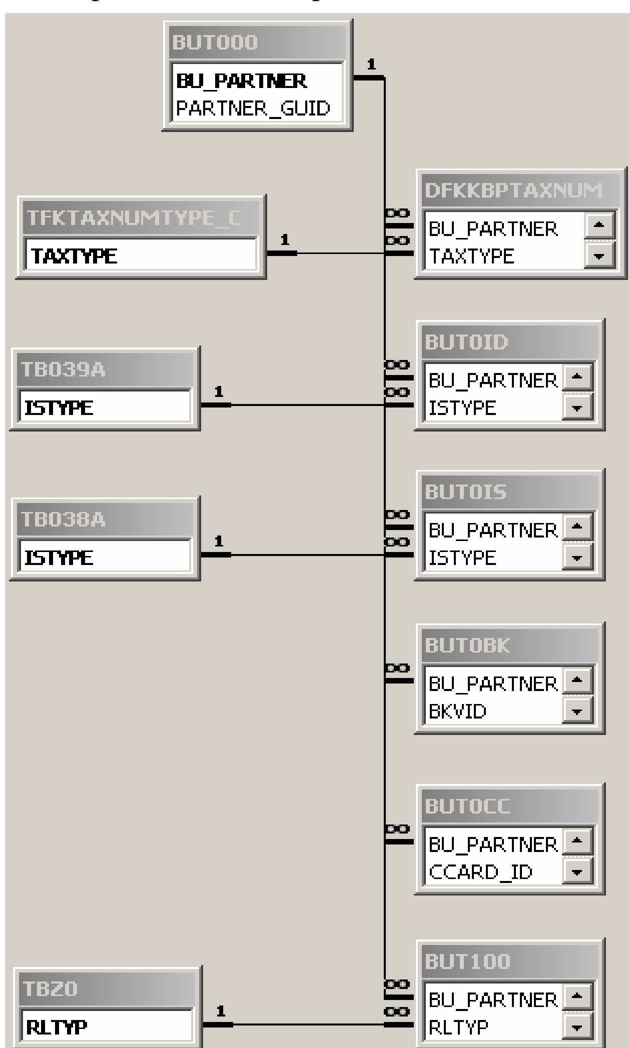
Handelt es sich bei der Geschäftspartnerbeziehung um eine Firmenbeteiligungsbeziehung sind in der BUT053 Tabelle folgende Daten hinterlegt: Prozentsatz der Kapitalbeteiligung, Betrag der Kapitalbeteiligung, und ein Währungsschlüssel. Ein Währungsschlüssel ist eine SAP interne Abkürzung für eine Währung.

Die Basistabelle der Geschäftspartnerbeziehungen BUT050 enthält zusätzlich Informationen, bzw. Datenfelder, die typunabhängig für die beiden Ausprägungen BUT051 und BUT053 gelten. Es handelt sich dabei um betriebswirtschaftliche nicht relevante Informationen wie: Benutzer- und Datumsangaben, wann ein Beziehungsobjekt angelegt und das letzte Mal geändert wurde.

Die Art der Generalisierung von BUT051 und BUT053 gegenüber BUT050 ist zum einen disjunkt, da nur BUT051 und BUT053 von einem Subtyp BUT050 sein können. Die Generalisierung ist auch

vollständig, weil sowohl BUT051 als auch BUT053 auf jeden Fall dem einen Basistyp zugeordnet werden müssen. Realisiert wird diese Generalisierungsbeziegung mit dem Primärschlüsselattribut BUT050::RELNR, der Geschäftspartnerbeziehungsnummer, das auch die beiden Subtyen besitzen.

Jeder Geschäftspartnerbeziehung ist eine Geschäftspartnerbeziehungsadresse BUT052 zugeordnet. Dazu besitzt BUT052 die Fremdschlüsselattribute ADDRNUMBER und RELNR. RELNR verweist auf BUT050::RELNR, BUT052::ADDRNUMBER auf BUT020::ADDRNUMBER. Die BUT020 Tabelle bildet eine Geschäftspartneradresse ab. Eine Geschäftspartneradresse kann mehrere Geschäftspartneradress**verwendungen** haben. Hierfür besteht eine Fremdschlüsselverbindung zwischen BUT020::ADDRNUMBER und BUT021::ADDRNUMBER. BUT021 ist der Verbindungsknoten zum Customizing, denn BUT021 ist mittels BUT021::ADR_KIND mit der Customizing Tabelle TB009 verknüpft, wo Geschäftspartneradressarten individuell definiert werden können.



Zum betriebswirtschaftlichen Umfeld aller Typen von Geschäftspartnern gehören die Informationen in den Tabellen, die in der nebenstehenden Abbildung eine 1:n Beziehung zur Geschäftspartnertabelle BUT000 haben. Allen ist gemeinsam, dass ihnen ebenfalls per 1:n Verknüpfung eine Verbindung zu Customizingtabellen zugedacht ist. Zu diesem Umfeld gehören:

- **DFKKBPTAXNUM**
  Enthält eine Steuernummer für einen Geschäftspartner und hat mit DFKKBPTAXNUM::TAXTYPE eine Verbindung zu einer Customizingtabelle für einen Steuernummerntyp. Die Bezeichnung für einen Steuernummerntyp indentifiziert geographisch den Sitz eines Unternehmens. TAXTYPE ist z.B. für Frankreich: SIREN oder für Polen NIP.
- **BUT0ID**
  Dient zur zusätzlichen eindeutigen Identifizierung eines Geschäftspartners. Hierfür besitzt diese Tabelle das Attribut IDNUMBER, mehrere Datumsfelder, sowie eine Verbindung zur Customizing Tabelle für die Identifikationsart analog zum Steuernummerntyp der Steuernummerntabelle.

Abbildung 14: Umgebungstabellen des Geschäftspartners mit ausgelagerten Daten zur Verbindung mit den Customizing Tabellen.

- **BUT0IS**
  Jeder Geschäftspartner wird einer Branche zugeordnet. Hierfür gibt es das Feld BUT0IS:: IND_SECTOR für den Namen order den Identifier der Branche. Für die Verbindung zur Customizing Tabelle TB039A für das Branchensystem gibt es das Fremdschlüsselattribut BUT0IS::ISTYPE.

- **BUT0BK**
  Eine Geschäftspartnerbankverbindung. Sie enthält neben einer Bankverbindungsidentifikationsnummer BUT0BK::BKVID eine Datenstruktur mit Attributen wie Kontonummer und den Namen des Kontoinhabers und der Bankverbindung.
- **BUT0CC**
  Bildet Informationen über eine Zahlungskarte ab. Neben dem Primärschlüsselfeld BUT0CC::CCARD_ID, beinhaltet eine interne Datenstruktur Daten wie eine Kartennummer, eine Bezeichnung für die Zahlungskartenverbindung und eine Zahlungskartenart.
- **BUT100**
  Immer von besonderer Bedeutung für jede Art und Funktion eines SAP System Benutzers und auch eines Geschäftspartners ist seine Rolle, die er einnimmt. BUT100 ist dabei eine reine Fremdschlüsselzuordnungstabelle zur Customizing Tabelle TB008 mit dem Fremdschlüssel TB008::RLTYP, bzw. BUT100::RLTYP.

Bisher wurde noch nicht behandelt, wo personenbezogene Daten und Adressinformationen hinterlegt sind. Die Informationsträger dieser Daten sind die Tabellen ADRC und ADRP. Die Verknüpfungspunkte zum Zugriff auf die Inhalte dieser beiden Tabellen sind die Fremdschlüsselattribute der Extension Tabellen BUT052, BUT020 und BUT000. Die bereits betrachteten Tabellen BUT052 und BUT020 sind reine n:m Beziehungsauflösungstabellen mit BUT052 zwischen BUT050 und ADRC, sowie mit BUT020 zwischen BUT000 und ADRC. Das in beiden, BUT052 und BUT020, notwendige Fremdschlüsselattribut ist ADRC::ADDRNUMBER.

Die Tabelle BUT000 besitzt ein Feld BUT000::PERSNUMBER. Handelt es sich bei einem Geschäftspartner um den Generalisierungstyp Natürliche Person verweist der Wert dieses Attributs auf einen Datensatz in der Tabelle Person ADRP. Dessen Primärschlüsselfeld ist nämlich ADRP::PERSNUMBER. So stehen also jedem Datensatz in BUT000, sofern BUT000::TYPE den Wert 1 für Natürliche Person hat, die zugehörigen persönlichen Daten des Geschäftspartners zur Verfügung.

Das folgende Schaubild zeigt die Verknüpfungen der Adressdaten Entitäten über Verknüpfungstabellen mit der Geschäftspartner Tabelle BUT000.

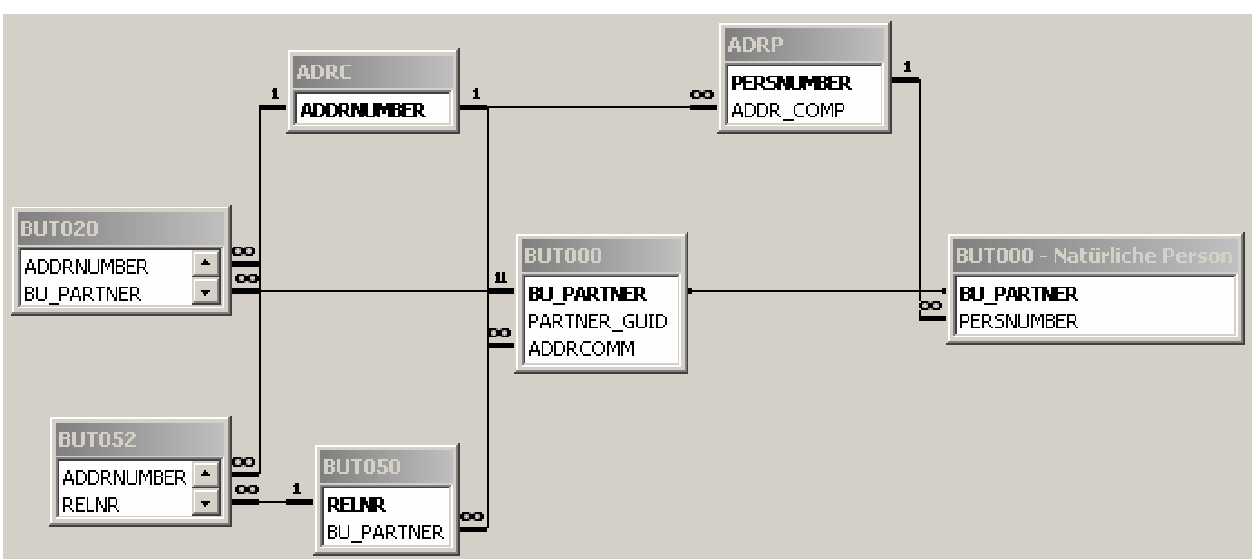


Abbildung 15: Verbindung von Adressdatenentitäten zur Geschäfstpartnerhaupttabelle BUT000

3.3.2 Customizing - Tabellen des Geschäftspartners

Dieser Abschnitt beschreibt den Unternehmens- bzw. auch branchenabhängigen Teil des Geschäftspartners, für den SAP nur das Aussehen der Schnittstellen definiert, sprich die Bezeichnung der Tabellen und deren Verknüpfung zu den richtigen Tabellen des strategischen Bereichs des Geschäftspartners. Die Customizing Tabellen des Geschäftspartners sind schlank, d.h. sie besitzen ausser den Fremd- und Primärschlüsselattributen meistens nicht mehr als ein Feld, das mit unternehmensspezifischen Daten gepflegt werden muss.

Für den Geschäftspartner gibt es die folgenden Customizing Tabellen, die sich in verschiedene Kategorien einordnen lassen:

Kategorie	Tabellenname	Tabellenbezeichnung
GESCHÄFTSPARTNERROLLE	TB008	Geschäftspartner Rolle
	CRMC_BUPA_ RLTPRP	Geschäftspartnerrolle Funktionsvorschlags-schema
	CRMC_BUPA_ PRPS	Funktionsvorschlagsschemata für Partner-funktionen
	CRMC_BUPA_ PRPFCT	Zuordnung: Partnerfunktionen Funktionsvorschlagsschema
	CRMC_PARTNER_ FCT	Partnerfunktionen
	CRMC_PARTNER_ PFT	Partnerfunktionstypen
GESCHÄFTSPARTNERGRUPPEN-HIERARCHIETYP	BUT_HIER_TYPE	Geschäftspartnergruppen Hierarchietyp
GESCHÄFTSPARTNERBEZIE-HUNGSART	TBZ9	Geschäftspartner Beziehungstyp
	TB905	Geschäftspartner Beziehungsart
GESCHÄFTSPARTNERBEZIE-HUNGSATTRIBUT	TB910	Ansprechpartner Abteilungsart
	TB912	Ansprechpartner Funktion
	TB914	Ansprechpartner Vollmacht
	TB916	Ansprechpartner VIP Kennzeichen
GESCHÄFTSPARTNERATTRIBUT	TB009	Geschäftspartner Adressart
	TB008S	Adressfindungsvorgang
	TB008U	Adressfindungsvorgangsart
	TBZ3P	Geschäftspartnertyp
	TB004	Geschäftspartnerart
	TB001	Geschäftspartnergruppierung
	TB005	Geschäftspartner Datenherkunftsart
	TFKTAXNUMTYPE	Geschäftspartner Steuernummertyp
	TFKTAXNUMTYPE_C	Geschäftspartner Steuernummertypauswahl
GESCHÄFTSPARTNERPERSONEN-ATTRIBUT	USR02	Anwender
	TB027	Familienstand
	TB028	Geschäftspartnerberuf
GESCHÄFTSPARTNERGRUPPENART	TB025	Geschäftspartner Gruppenart
GESCHÄFTSPARTNER ORGANISA-TIONSATTRIBUT	TB019	Geschäftspartner Rechtsform
	TB032	Rechtsträger
GESCHÄFTSPARTNER IDENTIFI-	TB039	Geschäftspartner Identifikationstyp

KATIONSART	TB039A	Geschäftspartner Identifikationsart
BRANCHENKATALOG	TB038	Branchenkatalog
	TB038A	Branche
BERRECHTIGUNGSGRUPPEN-OBJEKT	TB036	Berechtigungsgruppenobjekt
	TB037	Berechtigungsgruppe

Abbildung 16: SAP CRM Customizing Tabellen des Geschäftspartners

GESCHÄFTSPARTNERROLLE

In der Kategorie Geschäftspartnerrolle spielen die beiden Customizing Tabellen TB008 und CRMC_BUPA_RLTPRP mit der Extensiontabelle BUT100 zusammen, um einem Geschäftspartner passende Rollen zuzuordnen. Eine Rolle ist definiert als „betreibswirtschaftlicher Kontext eines Geschäftspartners", auf Seite 22 in [8]. D.h. Eine Rolle weist einem Geschäftspartner betriebswirtschaftliche und rechtliche Funktionen zu, die er bei seiner Geschäftstätigkeit wahrnimmt. Ein Geschäftspartner kann mehrere dieser Rollen besitzen und einer Rolle können mehrere Funktionen oder Rechte zugeordnet sein. Es gibt auf der einen Seite von SAP vordefinierte Rollen und zugeordnete Funktionen. Auf der anderen Seite die Möglichkeit selbst neue Rollen zu definieren, sowie diesen entweder ebenfalls eigens erstellte Funktionen zuzuordnen oder auf vordefinierte, von SAP gegebene Funktionen zurückzugreifen.

Das folgende Schaubild zeigt diesen Zusammenhang mittels der Datenbanktabellen, die SAP hierfür vorsieht:

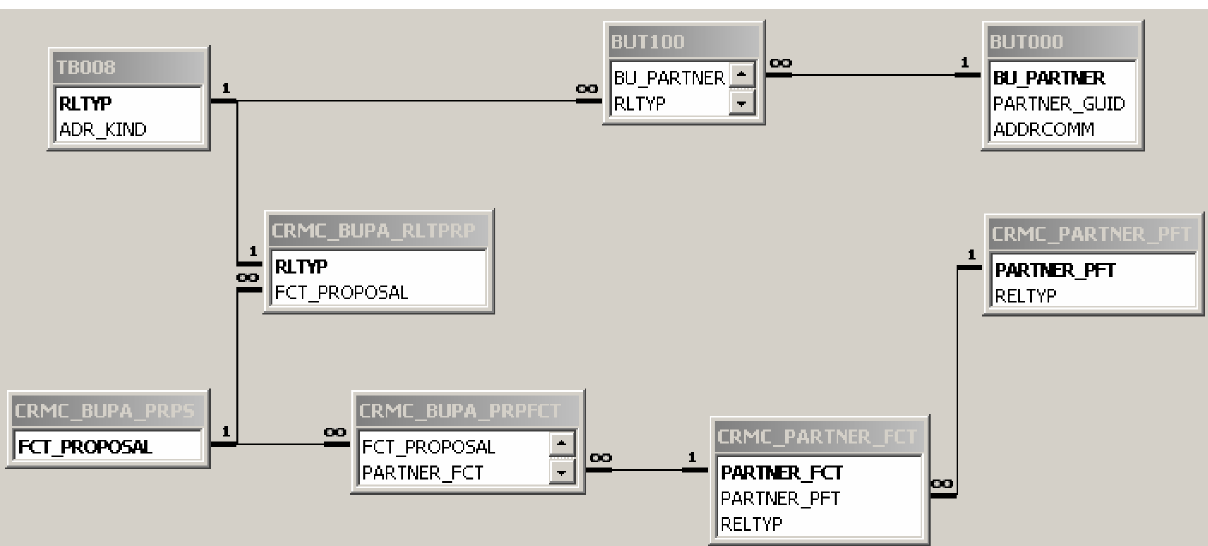


Abbildung 17: Beziehungskonstrukt zwischen Geschäftspartnerhaupttabelle und Geschäftspartnerrollen

Die Tabelle BUT100 ist, als Rollenzuordnungstabelle der Knotenpunkt zwischen dem Geschäftspartner BUT000 und der Geschäftspartnerrolle TB008, also auch zwischen Customizing und der strategiscvhen Sicht auf den Geschäftspartner. Einer Rolle werden Funktionen, nicht direkt zugeordnet, sondern zwischen den Rollen und den Funktionen stehen Funktionsvorschlagsschemata. Einer Rolle ist genau eine Funktionsvorschlagsschemazuordnung CRMR_ BUPA_RLTPRP zugeordnet, die wiederum einer Rolle mehrere Funktionsvorschlagsschemata CRMC_BUPA_PRPS zuweist. Die Verbindung von den Funktionsvorschlagsschemata zu den eigentlichen Funktionen leistet die Tabelle CRMC_BUPA_PRPFCT. Als reine n:m Beziehungsauflösungstabelle mit den Schlüsselattributen CRMC_BUPA_PRPS::FCT_PROPOSAL und CRMC_

BUPA_PRPFCT::PARTNER_FCT können einem Funktionsvorschlagsschema mehere Funktionen, sprich CRMC_PARTNER_FCT Datensätze zugeordnet sein.

Schließlich gehört jede Funktion zu einem Funktionstyp CRMC_PARTNER_PFT. Da zu einem Funktionstyp mehrere Funktionen gehören können, besteht zwischen CRMC_PARTNER_FCT::PARTNER_PFT und CRMC_PARTNER_PFT::PARTNER_PFT eine 1:n Beziehung.

GESCHÄFTSPARTNERGRUPPENHIERARCHIETYP

Das Customizing für Geschäftspartnergruppenhierarchie**typen** ist im Vergleich zu den Geschäftspartnerrollen einfach. Es muss nur die Tabelle BUT_HIER_TYPE und dessen Attribut HIERARCHY_TYPE gepflegt werden. Diese Tabelle steht mit keiner weiteren in einer referenziellen Beziehung.

GESCHÄFTSPARTNERBEZIEHUNGSART

Die Geschäftspartnerbeziehungsart ist ein Teil des Customizing, der mit der Geschäftspartnerrolle in Verbindung steht. D.h. genauer: jeder Geschäftspartnerbeziehungsart wird eine Rolle zugeordnet. Diese Verbindung wird mittels TB008::RLTYP und TBZ9::RLTP1 realisiert. Das folgende Schaubild verdeutlicht das Customizing der Geschäftspartnerbeziehungsart und wiederholt nochmals die Verbindung zur Geschäftspartner –und Geschäftspartnerbeziehungstabelle.

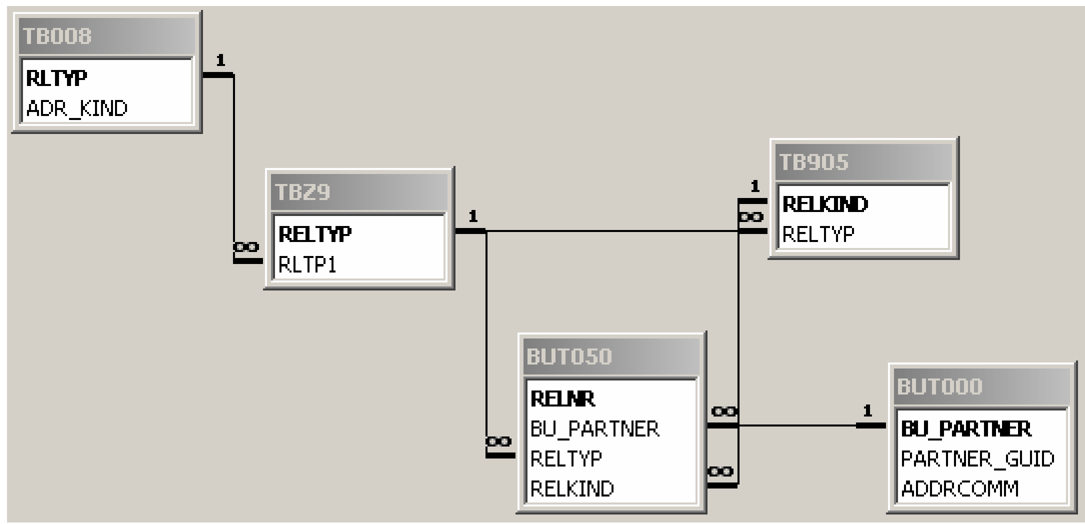


Abbildung 18: Zusammenhang von Geschäftspartnerbeziehungsart und Geschäftspartnerrolle

GESCHÄFTSPARTNERBEZIEHUNGSATTRIBUT

Die vier Customizing Tabellen dieser Kategorie dienen zum Abbilden von unternehmens- und branchenspezifischen Details einer Ansprechpartnerbeziehung BUT051, also einer Unterart der Geschäftspartnerbeziehung.

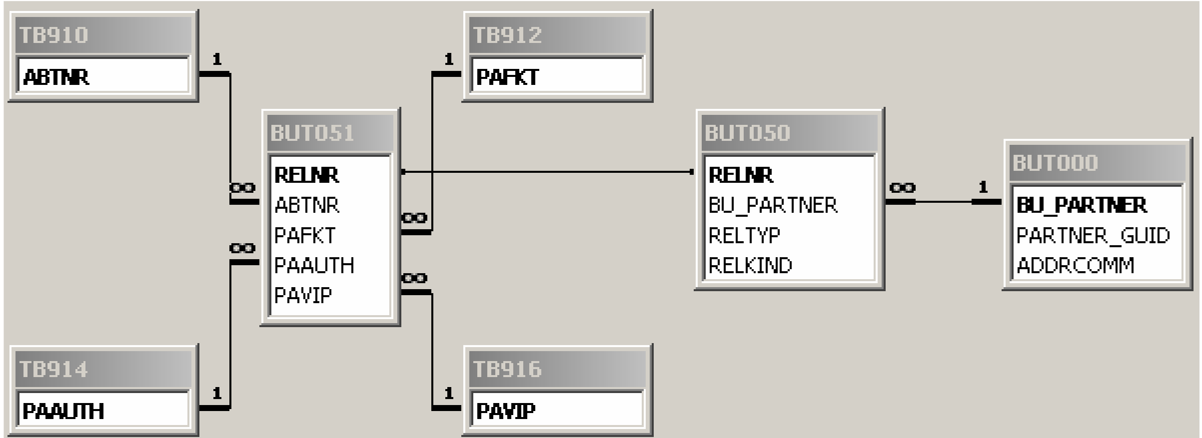


Abbildung 19: Die Ansprechpartnerbeziehung eines Geschäftspartners

Jedem Datensatz in BUT051 ist per Fremdschlüsselattribut eine Ansprechpartnerabteilungsart TB910, Ansprechpartnerfunktion TB912, Ansprechpartnervollmacht TB914 und ein Ansprechpartner VIP Kenzeichen TB916 zugeordnet.

GESCHÄFTSPARTNERATTRIBUT

Jedem Geschäftspartner egal welchen Typs, werden unternehmensspezifizierbare Details zugewiesen. Die Beziehungen zur Geschäftspartner Tabelle BUT000 sind analog zu denen zwischen BUT051 und TB91x aus der Kategorie Geschäftspartnerbeziehungsattribut. D.h. die üblichen 1:n Kardinalitäten mit dem Unterschied, dass die 1:n Beziehung in die entgegengesetzte Richtung zeigt im Vergleich zum Verhältnis der Extension Tabellen zum Geschäftspartner BUT000. Dies bedeutet, dass in der BUT000 Tabelle die Fremdschlüssel zu den Customizing Tabellen hinterlegt sind. Zu den Geschäftspartnerattributen gehören:

- Adressart → TB009
- Geschäftspartnertyp → TBZ3P
- Geschäftspartnerart → TB004
- Geschäftspartnergruppierung → TB001
- Datenherkunftsart → TB005

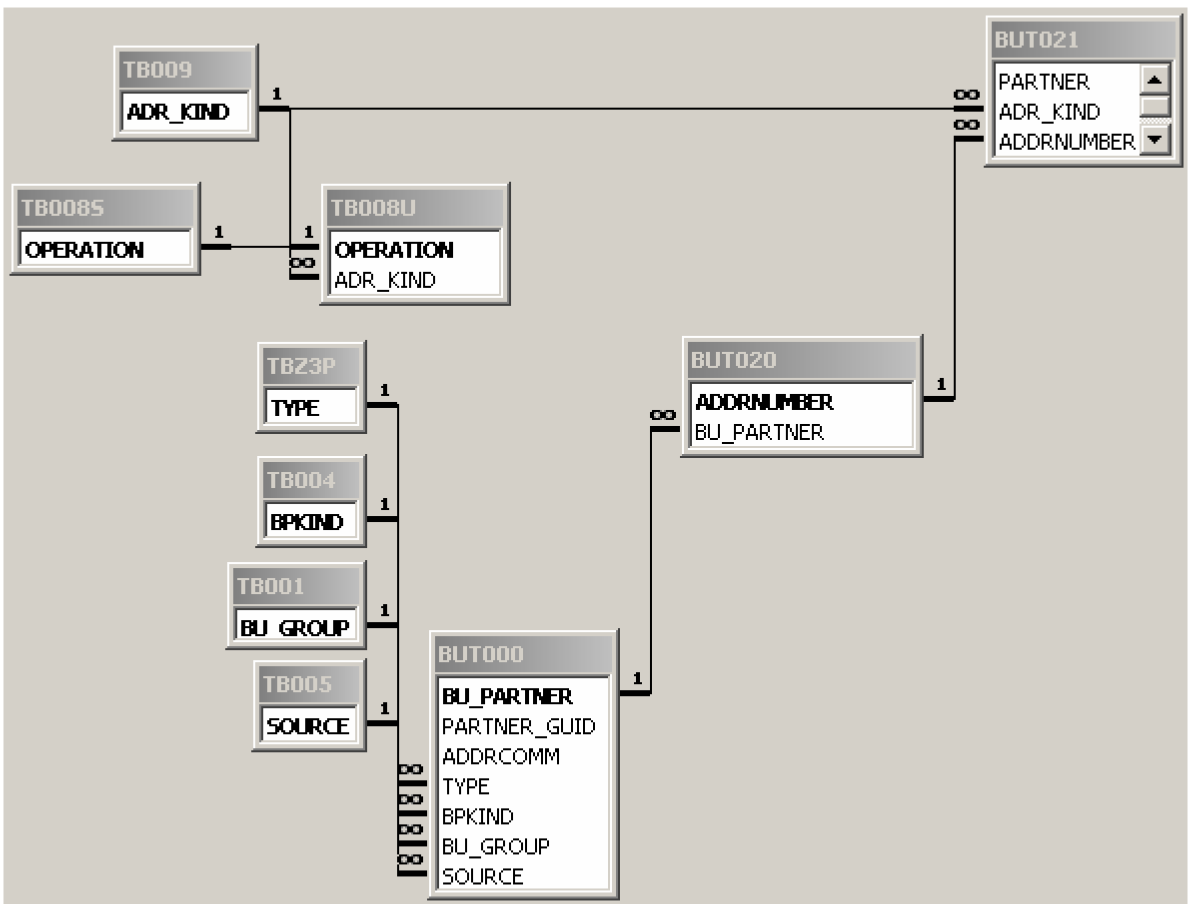


Abbildung 20: Geschäftspartnerattribute

Das Customizing der Geschäftspartneradressart muss gesondert betrachtet werden. Dies wird nicht direkt als Verbindung zur BUT000 Tabelle abgebildet, sondern geschieht über Nicht-Customizing Tabellen BUT021 und BUT020. Jeder Geschäftspartner besitzt eine Geschäftspartner-adresse. Realisiert wird dies mit BUT000::BU_PARTNER und BUT020::BU_PARTNER. Jeder Geschäfts-partneradresse besitzt eine Geschäftspartneradressverwendung. Also besteht eine 1:n Beziehung zwischen BUT020::ADDRNUMBER und BUT021::ADDRNUMBER.

Die Verbindungsknoten zum Customizing ist die Tabelle BUT021. Denn sie besitzt das Fremd-schlüsselfeld ADDR_KIND, was ihr eine Geschäftspartneradressart TB009 zuordnet. In dieser Eigenschaft ist sie eine n:m Auflösungstabelle zwischen BUT021 und TB009. D.h. insgesamt: jeder Geschäftspartner hat eine oder mehrere Geschäftspartneradressen. Jede Geschäftspartneradresse kann mehrere Adressverwendungen haben. Ein betriebswirtschaftlicher Anwendungesfall kann so aussehen, dass eine Geschäftspartneradresse für den Warenempfang und als Lieferadresse *verwendet* werden kann. Somit hat eine Geschäftspartneradresse mehere Adressarten und jede Adressart kann natürlich für mehrere Adressen gelten. Die Definition der Adressarten ist hierbei eine Aufgabe der Customizing Tätigkeit.

Damit im Zusammenhang steht der Adressfindungsvorgang. Dies kann z.B. lauten: „Bestel-lung senden“ oder „Waren empfangen“. Jede Adressart kann mehrere dieser Findungsvorgänge besitzen, aber ein Findungsvorgang gehört nur zu einer Adressart.

GESCHÄFTSPARTNER PERSONENATTRIBUT

Jedem Geschäftspartner des Typs Natürliche Person sind die Attribute Anwender USR02, mit SAP Anmeldedaten des Anwenders, Familienstand TB027 und Geschäftspartnerberuf TB028 zugeordnet.

GESCHÄFTSPARTNERGRUPPENART

Jedem Geschäftspartner des Typs Geschäftspartnergruppe wird genau eine Geschäftspartnergruppenart TB025 zugewiesen. Wie diese Gruppenarten heißen, genauer gesagt, welche Werte das Gruppenartattribut TB025::PARTGRPTYP hat, obliegt dem Customizing.

GESCHÄFTSPARTNER ORGANISATIONSATTRIBUT

Jedem Geschäftspartner des Typs Organisation stehen genau ein Rechtsträger TB032 und eine Rechtsform TB019 zur Verfügung. Im Customizing muss hier das Feld LEGAL_ORG von TB032 gepflegt werden. Für die Rechtsform ist dies das Attribut LEGAL_ENTY.

GESCHÄFTSPARTNER IDNENTIFIKATIONSART

Der Extension Tabelle BUT0ID für die Identifikation des Geschäftspartners ist genau eine Geschäftspartneridentifikationsart zugeordnet. Zu konfigurieren ist TB039A::TYPE. Z.B. kann es als Identifikationsarten Reisepass oder Personalausweis geben. Der Identifikationstyp TB039 ist nochmals eine Möglichkeit, Identifikationsarten zu kategorisieren. Hierzu besitzt TB039A das Fremdschlüsselattribut CATEGORY, das der Primärschlüssel der Tabelle TB039 ist. Wie diese Kategorien aussehen, ist im Customizing zu spezifizieren.

BRANCHENKATALOG

Die Customizing Tabellen für den Branchenkatalog sind nicht direkt mit der Geschäftspartnertabelle BUT000 verbunden, sondern über dessen Extension Tabelle für die Geschäftspartnerbranche BUT0IS. Jede Branche TB038A ist wie viele andere Entitäten in einem Ordnungsrahmen sortierbar, wie z.B. eine Geschäftspartnerhierarchie in eine Geschäftspartnerhierarchiegruppe. Für die Branche heisst dieser Ordnungsrahmen Branchenkatalog oder Branchensystem. Es ist beispielsweise sinnvoll, Branchen nach wirtschaftlichen Tätigkeiten in einen Branchenkatalog einzuordnen.

Für eine Branche muss als wichtigestes Attribut TB038A::IND_SECTOR gecustomized werden, was den Branchennamen abbildet. Die Zuordnung zu einem Branchenkatalog erfolgt über TB038A::ISTYPE, bzw. TB038::ISTYPE. Dies ist das Primärschlüsselfeld der Branchenkatalogtabelle TB038. In der Extension Tabelle BUT0IS wird als Verknüpfungsattribut für die Branche BUT0IS::IND_SECTOR verwendet. Zugleich zur Vereinfachung der Selektierung des zugehörigen Branchensystems ist auch dessen Primärschlüssel ISTYPE als Fremdschlüsselfeld hinterlegt.

BERECHTIGUNGSGRUPPENOBJEKT

Das Berechtigungsgruppenobjekt ist direkt dem Geschäftspartner zugewiesen. BUT000 besitzt ein Feld AUGRP, das auf den Primärschlüssel TB037::AUGRP der Tabelle Berechtigungs-gruppe verweist. Damit besitzt jeder Geschäftspartner egal welchen Typs immer genau eine Berechtigungsgruppe.

Ein Berechtigungsgruppenobjekt fasst mehrere Berechtigungsgruppen zusammen. Dazu besitzt jeder Berechtigungsgruppendatensatz einen Wert für den Fremdschlüssel TB037::AUOBJ für die Tabelle TB036.

3.4 Der Geschäftsvorgang

Den Kern des Geschäftsvorgangs bilden die Tabellen CRMD_ORDERADM_H für den Geschäftsvorgang allgemein und CRMD_ORDERADM_I für die Positionen, die der Geschäftsvorgang enthält. Ein Vorgang kann meherere Positionen enthalten, was die Kardinalität zwischen diesen beiden Entitäten festlegt: Eine 1:n Beziehung, d.h. dass jeder Vorgang n Positionen haben kann und jede Position genau zu einem Vorgang gehören muss. Für diese referentielle Integrität gibt es das Primärschlüsselattribut CRMD_ORDERADM_H::GUID, Fremdschlüssel für CRMD_ORDERADM_I::GUID.

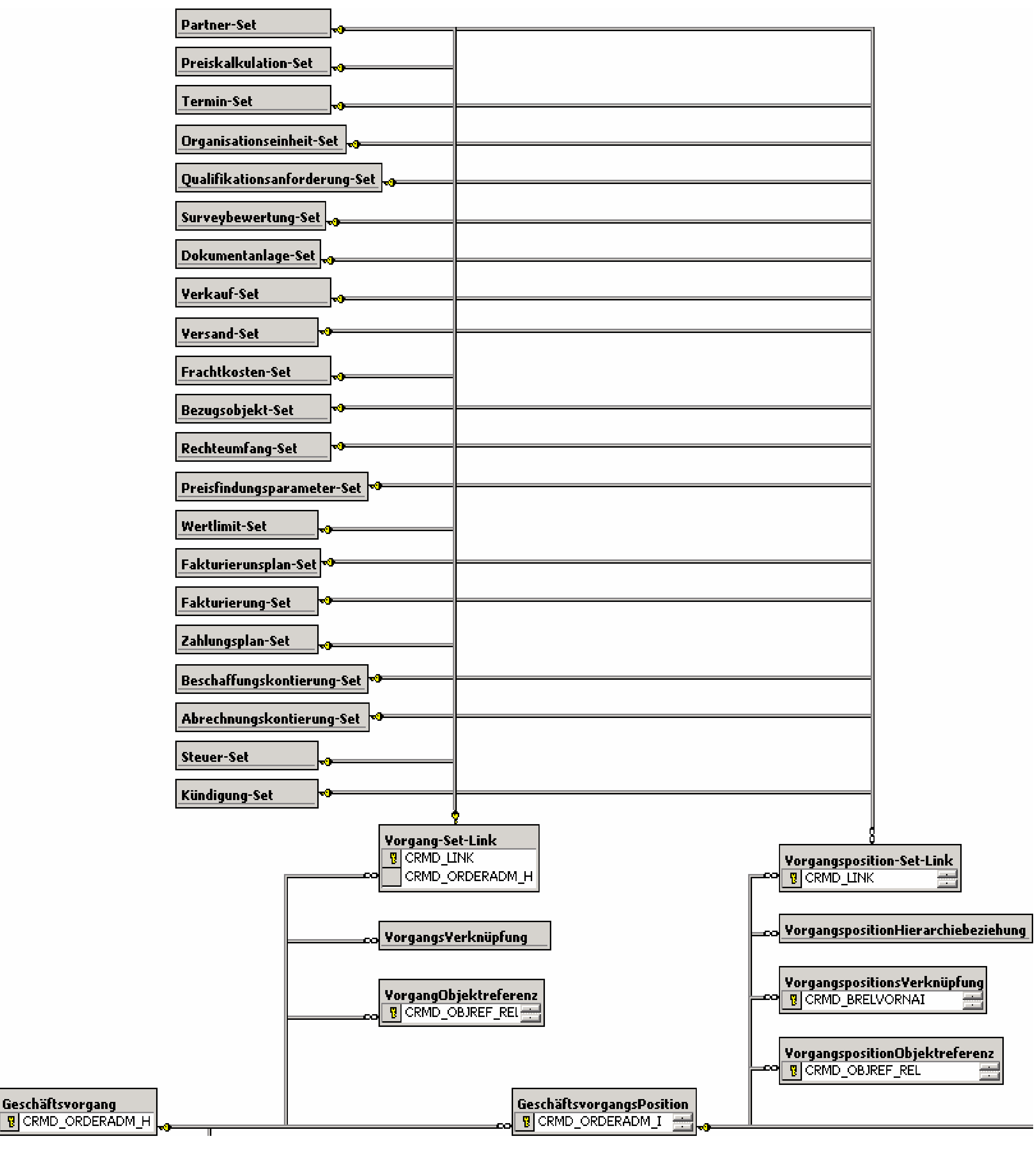
Es gibt eine große Vielzahl vorderfinierter Subtypen für den Geschäftsvorgang und dessen Positionen. Darüber hinaus können im Customizing weitere definiert werden. Die bedeutensten Subtypen sind der Geschäftsvorgangstyp Lead, Opportunity und Activity. Diese vordefinierten Subtypen unterscheiden sich von den meisten anderen durch das Vorhandensein einer Extratabelle. Andere Vorgänge wie z.B. die Reklamation oder alle Kontraktypen besitzen keine eigene Tabelle und werden durch das Attribut CRMD_ORDERADM_H::OBJECT_TYPE als Geschäftsvorgangstyp gekennzeichnet und unterschieden.

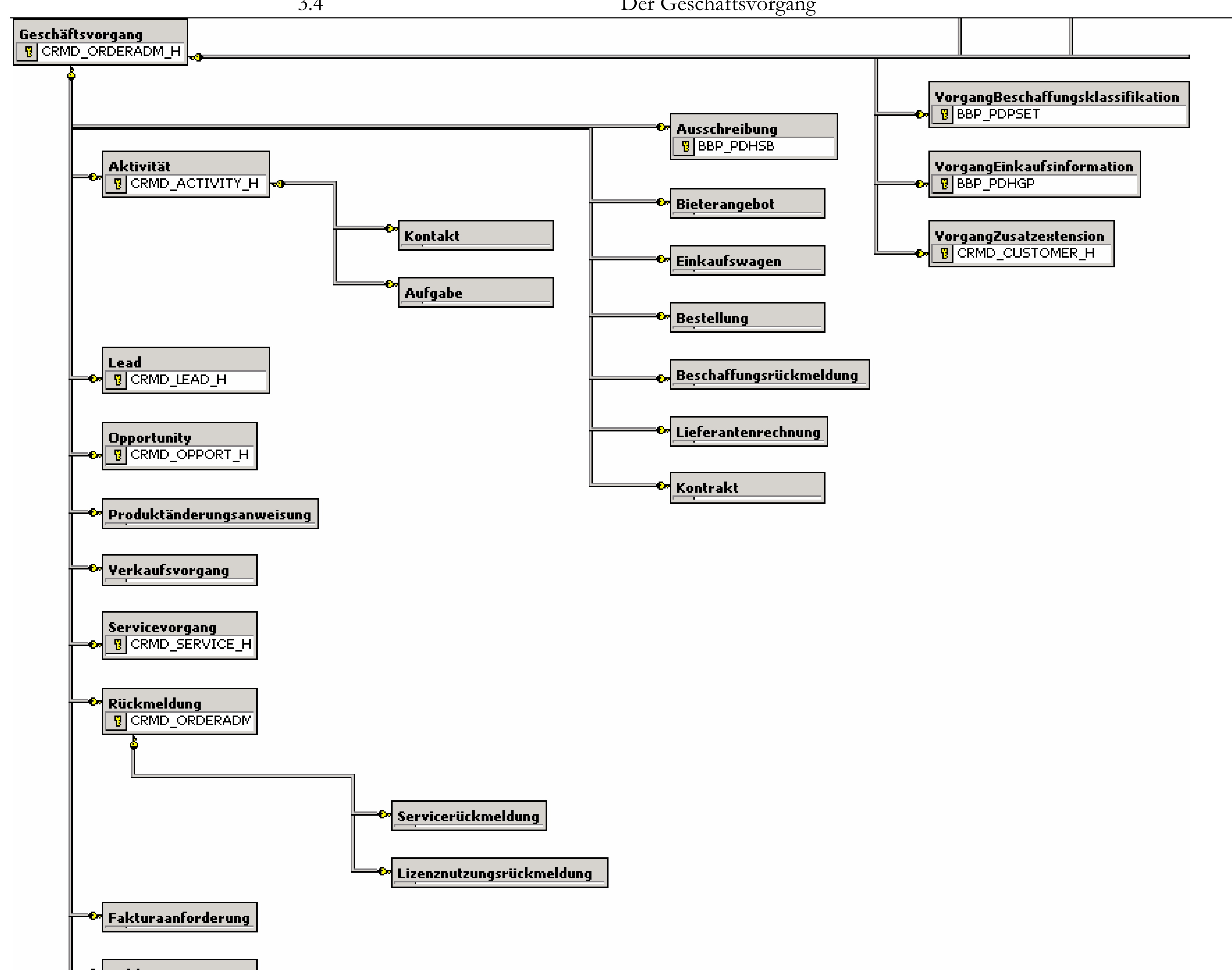
Da die CRMD_ORDERADM_H Tabelle Basistyp aller Geschäftsvorgangstypen ist, sind all deren Attribute für jeden Vorgangstyp relevant. Dies gilt auch für die diejenigen Vorgangstypen, die über das Attribut CRMD_ORDERADM_H::OBJECT_TYPE hinaus noch mit einer eigenen Tabelle repräsentiert sind. Für diese sind spezielle zusätzliche Eigenschaften definiert, wie z.B. für den Lead eine Leadpriorität. Wird ein neuer Lead angelegt, bearbeitet das SAP System zwei Tabellen. D.h. es erfolgt ein insert Statement über die Tabellen CRMD_ORDERADM_H und CRMD_LEAD_H.

Solch eine Generalisierungsbeziehung ist gleich der Situation in objektorientierten Programmiersprachen mit einfacher Vererbung. Auch dort beinhaltet eine Basisklasse alle für ihre abgeleiteten Klassen relevanten Eigenschaften. Für eine Subklasse mit zusätzlichen Feldern kann es nicht sein, dass ein Basisklassenattribut nicht ein Bestandteil seines Typs ist.

Anders als beim Geschäftspartner gibt es keine strategische Sicht auf den Geschäftsvorgang, sondern stattdessen die operative, die wiederum beim Geschäftspartner nicht vorhanden ist. Die operative Sicht des Geschäftsvorgangs besteht aus vielen Set Tabellen, die sowohl den Geschäftsvorgang, also auch dessen Positionen näher beschreiben.

Die Extension Tabellen, die den Geschäftsvorgang und dessen Positionen um betriebswirtschaftliche Gesichtspunkte erweitern, stehen in der üblichen 1:0…1 Beziehung. Die folgenden Tabellen stellen dar, welche Set-, bzw. Extensiontabellen es für den Geschäftsvorgang gibt. Daran anschließend folgt wie beim Geschäftspartner eine Analyse des Aufbaus und der Struktur, sprich der Verknüpfungen der Entitäten, die den Geschäftsvorgang ausmachen. Zunächst aber wie beim Geschäftspartner, ein Entity Relationship Modell des Geschäftsvorgangs.


Partner-Set
Preiskalkulation-Set
Termin-Set
Organisationseinheit-Set
Qualifikationsanforderung-Set
Surveybewertung-Set
Dokumentanlage-Set
Verkauf-Set
Versand-Set
Frachtkosten-Set
Bezugsobjekt-Set
Rechteumfang-Set
Preisfindungsparameter-Set
Wertlimit-Set
Fakturierunsplan-Set
Fakturierung-Set
Zahlungsplan-Set
Beschaffungskontierung-Set
Abrechnungskontierung-Set
Steuer-Set
Kündigung-Set
Vorgang-Set-Link
CRMD_LINK
CRMD_ORDERADM_H
VorgangsVerknüpfung
VorgangObjektreferenz
CRMD_OBJREF_REL
Vorgangsposition-Set-Link
CRMD_LINK
VorgangspositionHierarchiebeziehung
VorgangspositionsVerknüpfung
CRMD_BRELVORNAI
VorgangspositionObjektreferenz
CRMD_OBJREF_REL
Geschäftsvorgang
CRMD_ORDERADM_H
GeschäftsvorgangsPosition
CRMD_ORDERADM_I



Geschäftsvorgang
CRMD_ORDERADM_H
Ausschreibung
BBP_PDHSB
VorgangBeschaffungsklassifikation
BBP_PDPSET
VorgangEinkaufsinformation
BBP_PDHGP
VorgangZusatzextension
CRMD_CUSTOMER_H
Aktivität
CRMD_ACTIVITY_H
Kontakt
Aufgabe
Bieterangebot
Einkaufswagen
Bestellung
Beschaffungsrückmeldung
Lieferantenrechnung
Kontrakt
Lead
CRMD_LEAD_H
Opportunity
CRMD_OPPORT_H
Produktänderungsanweisung
Verkaufsvorgang
Servicevorgang
CRMD_SERVICE_H
Rückmeldung
CRMD_ORDERADM
Servicerückmeldung
Lizenznutzungsrückmeldung
Fakturaanforderung
Reklamation

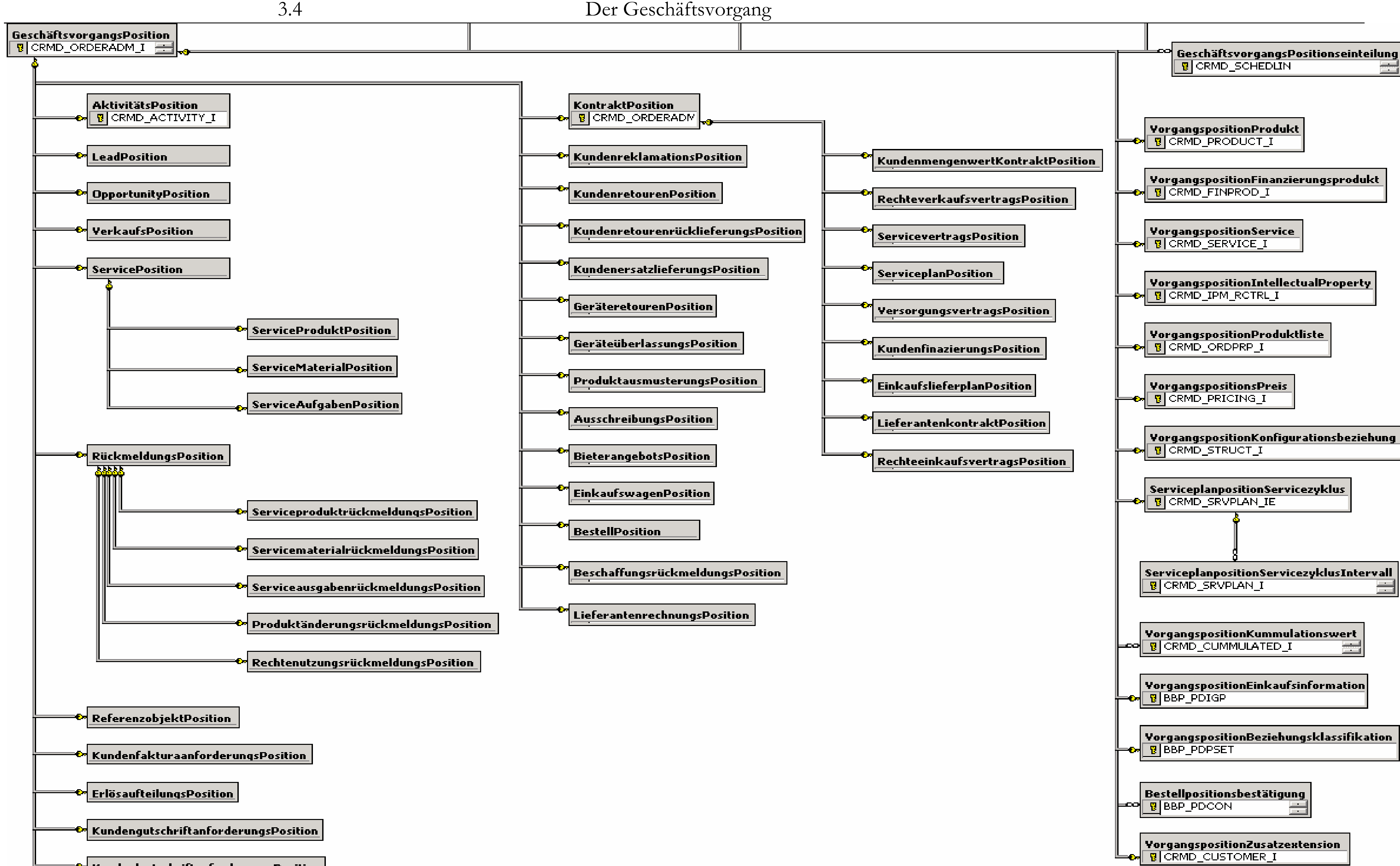


Abbildung 21: Das Entity Relationship Modell des SAP CRM Geschäftsvorgangs

3.4.1 Extensiontabellen des Geschäftsvorgangs und -positionen

Die folgende Tabelle listet alle Extensiontabellen des Geschäftsvorgangs und der Geschäftsvorgangspositionen auf. Es werden jeweils die ersten drei Tabellen von Vorgang und Position untersucht.

Entität	Tabellenname	Tabellenbezeichnung
GESCHÄFTSVORGANG	BBP_PDPSET	Vorgang Beschaffungsklassifikation
	BBP_PDHGP	Vorgang Einkaufsinformation
	CRMD_CUSTOMER_H	Vorgang Zusatzextension
GESCHÄFTSVORGANGSPOSITION	CRMD_PRODUCT_I	Vorgangsposition Produkt
	CRMD_FINPROD_I	Vorgangsposition Finanzierungsprodukt
	CRMD_SERVICE_I	Vorgangsposition Service
	CRMD_SRV_DEMAND	Vorgangsposition Service Ressourcenbedarf
	CRMD_IPM_RCTRL_I	Vorgangsposition Intellectual Property
	CRMD_ORDPRP_I	Vorgangsposition Produktliste
	CRMD_PRICING_I	Vorgangsposition Preis
	CRMD_STRUCT_I	Vorgangsposition Konfigurationsbeziehung
	CRMD_SRVPLAN_IE	Vorgangsposition Servicezyklus
	CRMD_SRVPLAN_I	Vorgangsposition Servicezyklusintervall
	BBP_PDIGP	Vorgangsposition Einkaufsinformation
	BBP_PDPSET	Vorgangsposition Beschaffungsklassifikation
	CRMD_CUSTOMER_I	Vorgangsposition Zusatzextension

Tabelle 9: Extensiontabellen des Geschäftsvorgangs

- **BBP_PDPSET**

Die Beschaffungsklassifikation für weitere Einkaufsinformationen besteht aus einem GUID Primärschlüsselfeld BBP_PDPSET::BBP_GUID und einer Reihe von boolschen Feldern, die angeben, ob für den zugehörigen Geschäftsvorgang Attribute, bzw. Bedingungen gegeben sind oder gegeben sein müssen. Ferner ob bestimmte Aktivitäten oder Aktionen ausgelöst werden sollen oder müssen.

Ein Geschäftsvorgang vom Typ Bestellung, Lieferantenrechnung oder Beschaffungsrückmeldung kann ein Wareneingangskennzeichen BBP_PDPSET::GR_IND, Rechnungseingangskennzeichen BBP_PDPSET::IR_IND, oder ein Kennzeichen für eine Rechnungsprüfung BBP_PDPSET:: GR_BASEDIV haben. Zusätzlich ist für diese drei Vorgangstypen ausgesagt, ob eine automatische Wareneingangsabrechnung erfolgen soll, eine Bestellbestätigung oder ein Lieferavis erwartet wird. All diese boolschen Attribute der BBP_PDPSET Tabelle sind in der Struktur BBP_PDS_PSET zusammengefasst.

- **BBP_PDHGP**

Diese Tabelle ist eine Sammlung von Beleginformationen und ist damit für finanzielle Transaktionen von Bedeutung. War die Extension BBP_PDPSET für die Geschäftsvorgangstypen Bestellung, Lieferantenrechnung oder Beschaffungsrückmeldung noch optional, ist die Erweiterung BBP_PDHGP verpflichtend. Realisiert wird diese Verbindung über CRMD_ORDERADM_H:: GUID und BBP_PDHGP::GUID. Zu den wichtigen Informationen dieser Vorgangsarten zählen:

- eine Referenzbelegnummer REF_DOC_NO
- eine Warenbegleitscheinnummer GR_GI_SLIP_NO
- eine Frachtbriefnummer BILL_OF_LADING
- einen Steuerbetrag TOTAL_TAX
- das Einkäuferkarteninstitut PCINS
- die Nummer der Einkäuferkarte PCNUM
- den Name des Einkäuferkarteninhabers PCNAME
- den Objekttyp des Vorlagebelegs SRC_OBJECT_TYPE
- die Identifikationsnummer eines Vorlagebelegs SRC_GUID
- die Bezug-Vorgangsnummer REF_OBJECT_ID
- ein Genehmigungskennzeichen APPROVAL_IND
- die Lieferzeit in Tagen DELIV_DAYS
- Ausprägung eines Einkaufsbelegs SUBTYPE
- Währung CURRENCY

All diese Informationen finden ebenfalls in den entsprechenden Papierbelegen Verwendung.

- **CRMD_CUSTOMER_H**

Der Name dieser Tabelle hat nichts mit einem Kunden- oder Geschäftspartner zu tun, die mit einem Geschäftsvorgang in Verbindung steht, sondern sie dient dazu, der Geschäftsvorgangs-entität einer SAP Installation individuelle Attribute hinzufügen zu können. Zur Verknüpfung zum Geschäftsvorgang dient das Feld CRMD_CUSTOMER_H::GUID.

Die Geschäftsvorgangsposition besitzt mehr Extensions im Vergleich zum ihr zugeordneten Geschäftsvorgang. Das liegt daran, dass einer Position ein materielles Produkt oder eine Dienstleistung, ein sogenanntes Intellectual Property zugeordnet ist. Verbunden mit jeglicher Art von Produkt oder Dienstleistung sind immer finanzielle Transaktionen. Daher besitzen einige der Extensionen Tabellen entweder direkte Felder, die für diese finanziellen Transaktionen relevant sind, oder Schlüsselattribute, die auf andere Tabellen mit dieser Art von Information verweisen.

Die nun folgenden Betrachtungen der Extension Tabellen der Geschäftsvorgangsposition befassen sich zunächst mit dem wirtschaftlichen Gut, also dem materiellen oder immateriellen Produkt, das die Position beinhaltet. Die Tabellen und Attribute, die für die Buchung wichtige Daten bereitstellen, sind Gegenstand des zweiten Teils der Untersuchung. Zu den Tabellen, die das Produkt innerhalb der Position abbilden, gehören: CRMD_PRODUCT_I, CRMD_SERVICE_I, CRMD_IPM_RCTRL_I, CRMD_ORDPRP_I, CRMD_PDIGP, CRMD_PDPSET und CRMD_PDCON.

- **CRMD_PRODUCT_I**

Diese Tabelle enthält Attribute für Gewichtsangaben zu Produkten wie Bruttogewicht GROSS_WEIGHT, Nettogewicht NET_WEIGHT und eine Gewichtseinheit WEIGHT_UNIT. Des Weiteren beinhaltet sie Verbindungen zur CRM Komponente Produkt.

- **CRMD_FINPROD_I**

Es gibt nur eine Ausprägung einer Geschäftsvorgangsposition, die ein Finanzierungsprodukt beinhaltet. Die Informationsattribute für eine Kundenfinanzierungsposition dienen zur Errechnung einer Finanzierung für das Finanzierungsprodukt. Zu dieser Errechnung gibt es eine Vielzahl von Finanzierungsparametern und Flags, die mit in die Finanzierung einfließen oder nicht. Um einen Eindruck dieser Parameter und Flags zu gewinnen, sind die wichtigsten aufgelistet:

Parameter / Flag	Beschreibung
RESID_VALUE_FLAG	Restwertrisiko
RESID_VALUE_GRP	Restwertkurve
CLASSIFICATION	Vertragsklasse
LEAS_CLASS	Steuerliche Preisstrategie
TAX_FINANC_FLAG	Steuerfinanzierung erlaubt
CHRG_FIN_USE_FLG	Gebührenfinanzierung
COST_FIN_USE_FLG	Kostenweitergabe
SERV_FIN_USE_FLG	Servicefinanzierung
TAX_FIN_USE_FLG	Steuerfinanzierung
COST_FIN_USE_FLG	Kostenweitergabe
FMV_CURVE	Marktwertkurve
PRICING_FIMA	Ratenberechnungsverfahren
PTWACC_ROE	Eigenkapitalrendite vor Steuern
ATWACC_ROE	Eigenkapitalrendite nach Steuer
BUSINESS_PROCESS	Betriebswirtschaftlicher Vorgang: z.B.: Freigeben, Ablehnen, Genehmigen, Stornieren, Archivieren, Löschen, Bericht erstellen
LOOKUP_FOR_INTER	Zinsfindung: z.B.: Kundenzinssatz oder Leasinggeberzins
OPTION_TYPE	Optionsart: z.B.: vorzeitiger Kauf, Kauf zu Vertragsende, Anschlussfinanzierung, , Vertragsbeendigung, …
PRICING_PROCESS	Preisfindungsprozess: z.B.: vorzeitiger Kauf, Kauf, Konkurs, Beendigung einer Versicherung, Vertragsabschluss Accounting, …
INTCALC_M_EXT	Zinsberechnungsmethode. Beispiele: 360E/360, act/360, act365, act/actY (ISDA), 360E/365
IRR_M_EXT	Effektivzinsmethode. Beispiele: PAngV, AIBD/ISMA, Braess, Moosmüller, US-Methode, EU-act/365, EU-30,42/365, Linear
PAYMSCHED_CREATE	Methode zur Zahlungsplanerzeugung: z.B.: periodischer oder stichtagsbasierter Ausweis der Zinsen
BACKSOLVE_METHOD	Lösungsmethode. z.B.: Kundenzinssatz, Profit-Spread, Rate Buy Down, Zahlungen berechnen (+ Rate Buy Down/ Profit-Spread)

Tabelle 10: Felder der Tabelle CRMD_FINPROD_I

- **CRMD_SERVICE_I**

Für Vorgangspositionen, die als Produkt eine Dienstleistung enthalten. Die Tabelle dient als Grundlage für die Abrechnung benötigter zeitlicher Resourcen. So gibt es hierfür zwei wesentliche Felder: das Reaktionszeitschema CRMD_SERVICE_I::SRV_SERWI und das Bereitschaftsschema CRMD_SERVICE_I::SRV_ESCAL. Das Reaktionszeitschema definiert Termine für die Ausführung von Servicetätigkeiten. Das Bereitschaftsschema ist eine zeitliche Periode, für den Anspruch auf Serviceleistungen eines Kunden.

Die Informationen der Tabellenattribute CRMD_SERVICE_I::SERVICE_TYPE und CRMD_SERVICE_I::VALUATION_TYPE sind Grundlagen für die Preisfindung für die geleistete Servicedienstleistung der Geschäftsvorgangsposition. Das CRMD_SERVICE_I::SERVICE_TYPE Attribut bildet Servicearten ab, wie z.B. geleisteter Service während Überstunden, am Wochenende oder zur geregelten Arbeitszeit. Somit hat die Sericeart direkten Einfluss auf die finanzielle Abrechnung der geleisteten Arbeit wie die Bewertungsart CRMD_SERVICE_I::VALUATION_TYPE. Diese beschreibt die Resource Arbeitskraft näher. So kann die Serviceleistung von einer Fachkraft,

einem Ingenieur oder einem Spezialisten abgeleistet werden. Je nach Qualifizierung des Serviceleistenden kann ebenfalls die finanzielle Berechnung variieren. Die Themen Preisfindung, Preiskondition und Konditionstechnik sind eigenständige SAP Komponenten wie der Geschäftspartner und der Geschäftsvorgang.

Vorgängspositionen, die eine Dienst- oder Serviceleistung darstellen sind: Serviceaufgabenposition, Servicematerialposition, Serviceproduktposition und Serviceproduktrückmeldeposition.

3.4.2 Set Tabellen des Geschäftsvorgangs und dessen Positionen

Die Set Tabellen des Geschäftsvorgangs beinhalten zusätzliche ausgelagerte Informationen. Grundsätzlich gilt für Geschäftsvorgänge und deren Positionen, dass eine Position die Werte und Einstellungen aus einer Set Tabelle des übergeordneten Geschäftsvorgangs übernimmt, sofern die Position keinen Eintrag in der jeweiligen Set Tabelle hat. Ziel der Ausführungen zu den Set Tabellen ist es, einen Überblick über die externen Informationen zu einem Geschäftsvorgang, bzw. der Postionen zu geben und verständnisrelevante Details zu den einzelnen Set Tabellen zu vermitteln. Hierzu werden einige der vorhandenen Set Tabellen vorgestellt.

ABRECHNUNGSKONTIERUNGS-SET

Eine Abrechnungskontierung, sprich ein Datensatz in der Abrechnungskontierungs-Set Tabelle CRMD_AC_ASSIGN ist ein Ersatzkonstrukt für Geschäftsvorgänge, bzw. –positionen, für die keine Wertaufteilung in der Faktura erfolgt. Die wichtigsten Felder dieser Tabelle sind die CRMD_AC_ASSIGN::GUID für das Kontierungsobjekt, also der Vorgang oder der Position und der Abrechnungskontierungsobjekttyp CRMD_AC_ASSIGN::AC_OBJECT_TYPE. Sollen beispielsweise vier Geschäftsvorgangspositionen getrennt voneinander abgerechnet werden, so wird für jede Position eine Abrechnungskontierung erstellt.

BESCHAFFUNGSKONTIERUNGS-SET

Die Felder der Set Tabelle BBP_PDACC sichern alle Informationen für die Verbuchung eines Geschäftsvorgangs in einem Sachkonto, einer Kostenstelle oder einem Geschäftsbereich. Relational verknüpft ist das Beschaffungskontierungs-Set über das BBP_PDACC::SET_GUID Attribut mit der Tabelle BBP_PDACC::CRMD_LINK und dessen Schlüsselattribut. Die folgende List beinhaltet ein paar der Felder, die Informationen abbilden, die für eine Kontierung von Bedeutung sind:

Feldname	Beschreibung
G_L_ACCT	Nummer des Sachkontos
BUS_AREA	Geschäftsbereich
COST_CTR	Kostenstelle
SD_DOC	Vertriebsbelegnummer
SDOC_ITEM	Verkaufsbelegposition
ORDER_NO	Auftragsnummer
CO_AREA	Kostenrechnungskreis
PROFIT_CTR	Profit Center
ACTIVITY	Vorgangsnummer
PART_ACCT	Kundennummer des Partners
COST_OBJ	Kostenträger

CMMT_ITEM	Finanzposition
FUNDS_CTR	Finanzstelle

Tabelle 11: Felder der Tabelle BBP_PDACC

Dem Geschäftsvorgangstyp Beschaffungsrückmeldungsposition muss ein Beschaffungs-kontierungs-Set zugeordnet werden. Optional *kann* einer Bestellung dieses zugewiesen werden.

Dokumentanlage-Set

Die Tabelle BBP_PDATT enthält ein oder mehrere Dokumentanlagen zu einem Geschäftsvorgang oder einer Position. Die wesentlichen Informationen zu einer Dokumentenanlage werden in den folgenden Feldern abgelegt:

Feldname	Beschreibung
OBJKEY	Objektschlüssel
DOC_ID	Dokument-ID
DOC_VER_NO	Versionsnummer des Dokuments
DOC_VAR_ID	Varianten ID des Dokuments
URL	Webadresse zum Dokument
TYPE	Dokumententyp. Z.B. ppt, pdf, doc,…
LOIO_CLASS	Dokumentenklasse

Tabelle 12: Felder der Tabelle BBP_PDATT

Folgende Vorgangstypen können eine Zuordnung zu diesem Set besitzen:

- Ausschreibung
- Beschaffungsrückmeldung
- Bestellung
- Bieterangebot
- Lieferantenrechnung

Fakturierungs-Set

CRMD_BILLING ist die Haupttabelle für alle Fakturainformationen eines Geschäftsvorgangs oder einer Position. Welche Informationen dies im Einzelnen sind listet die folgende Tabelle auf.

Feldname	Beschreibung
OBJKEY	Objektschlüssel
PERIOD_DATE	Periodendatum
RULE_PERIOD_DATE	Regel zur Ermittlung des Periodendatums
SETTL_FROM	Abrechnungsbeginn der Periode
RULE_SETTL_FROM	Regel zur Ermittlung des Abrechnungsbeginns der Periode
SETTL_TO	Abrechnungsende der Periode
RULE_SETTL_TO	Regel zur Ermittlung des Abrechnungsendes der Periode
BILL_DATE	Fakturadatum für Fakturaindex und Druck
RULE_BILL_DATE	Regel Herkunft nächstes Fakturierungsdatum
INVCR_DATE	Fakturaerstellungsdatum

RULE_INVCR_DATE	Regel Herkunft nächstes Fakturaerstellungsdatum
BILLING_BLOCK	Fakturasperrgrund
BUAG_GUID	Schlüsselattribut einer Geschäftsvereinbarung
SETTL_FIMA	Internes FIMA-Fakturierungsdatum
FIMA_KONDIND	Referenz FIMA-Konditionsposition
FIMA_BEWEIND	Referenz FIMA-Bewegungssatz

Tabelle 13: Felder der Tabelle CRMD_BILLING

Die Fakturainformationen von CRMD_BILLING beinhalten Datums sowie deren Ermittlungsregeln für den Abrechnungsbeginn, das Abrechnungsende, das Fakturadatum und das Fakturaerstellungsdatum. Der Fakturasperrgrund kann dazu verwendet werden, ein oder mehrere Positionen für die Faktura zu sperren. Ohne eine Beziehung zu einem Fakturierungs – Set kann eine Position nicht fakturiert werden.

Um über die Fakturierung die Finanzierung einer Position anzustoßen, besteht keine Beziehung zu Bankverbindungsinformationen. Diese ist über den Geschäftspartner verfügbar, der mit dem übergeordneten Geschäftsvorgang in Verbindung steht. In der Tabelle CRMD_ORDER_ INDEX werden alle Geschäftsvorgänge ihren Geschäftspartnern zugeordnet. Dies geschieht über die Attribute CRMD_ORDER_INDEX::HEADER für eine Geschäftsvorgangs GUID und CRMD_ ORDER_ INDEX::PARTNER_NO für die Identifikation des jeweiligen Geschäftspartners. Die Set Entitäten Zuordnungstabelle CRMD_LINK verwaltet, sofern angelegt einen Verweis auf einen Fakturierungs – Set Datensatz für einen Geschäftsvorgang. Der Geschäftspartner auf der anderen Seite besitzt Zugang zu den Bankverbindungsinformationen in der Tabelle BUT0BK. In dieser relationalen Beziehung ist festgelegt, dass ein Geschäftspartner ein oder mehrere Bankverbindungen haben kann.

Auf diese Weise kann einer über das Fakturierungs – Set zu fakturierenden Position über den Geschäftspartner immer eine Bankverbindung für die Fakturierung ermittelt werden.

FRACHTKOSTEN-SET

Das Frachtkosten – Set wird verwendet, um für den Geschäftsvorgangstyp Lieferantenrechnung, Frachtkosten für dessen Positionen zu verbuchen. Neben dem üblichen Verknüpfungsattribut BBPPDRFT::SET_GUID zur CRMD_LINK Tabelle sind für die Frachtkostenverbuchung folgende Felder relevant.

Feldname	Beschreibung
AMOUNT	Frachtkostenbetrag
TAX_CODE	Umsatzsteuerkennzeichen
CURRENCY	Währungsschlüssel

Tabelle 14: Felder der Tabelle BBP_PDACC

KÜNDIGUNG-SET

Das Kündigung Set besteht aus den beiden Tabellen CRMD_CANCEL und der Kündigungsindividualregel CRMD_CANCEL_IR. Einer Kündigung in CRMD_CANCEL können mehrere Kündigungsindividualregeln zugeordnet sein, weshalb zwischen den beiden Entitäten eine 1:n Beziehung besteht. Das Kündigung Set benutzen können ausschließlich alle Geschäftsvorgänge vom Typ Kontrakt. Es dient zur vorzeitigen Beendigung eines Kontrakts oder einer Position.

CRMD_CANCEL

Feldname	Beschreibung
CANCPROC	Kündigungsschema
CANCPARTY	Kündigungspartei
CANC_REASON	Kündigungsgrund

CRMD_CANCEL::GUID

CRMD_CANCEL_IR::CANC_GUID

CRMD_CANCEL_IR

Feldname	Beschreibung
CANCPERIOD_ID	ID der Terminregel der Kündigungsregel
CANCPERIOD_NAME	Name der Terminregel zur Kündigugsfrist
CANCRULE_T	Bezeichnung
CANCPARTY	Kündigungspartei
CANCDEADL_ID	ID der Terminregel (Frist zum Ausübungstermin)
CANCDEADL_NAME	Name der Terminregel (spätestes Eingangsdatum)
CANC_DURA	Wiederholrate
CANC_DURA_UNIT	Einheit der Dauer, in der sich die Wirks.termine wiederholen
CANC_NO_OF_EXEC	Anzahl der Termine

Abbildung 22: Beziehung der Tabellen CRMD_CANCEL und CRMD_CANCEL_IR

Das Kündigungspartei Tabellenattribut bildet die Personen ab, die an diesem Kontrakt beteiligt sind, und legt fest welche dieser Personen eine Kündigung aussprechen dürfen. Das Kündigungsschema ist eine Zusammenfassung von Kündigungsregeln, die wiederum Termine und Datumswerte beinhalten. Sind mehrere Kontrakte einem Kündigungsschema zugeordnet und werden an diesem Schema Änderungen vorgenommen, so wirkt sich dies auf alle Kontrakte aus, die auf dieses Kündigungsschema verweisen.

Eine Individualkündigungsregel, repräsentiert durch einen Datensatz der Tabelle CRMD_CANCEL_IR, kann nur genau einem Kontrakt zugewiesen sein. Sie beinhaltet sogenannte Terminregelversionen. Diese spiegeln die First CRMD_CANCEL_IR::CANCPERIOD_ID wider, in der gegündigt werden kann und die Gültigkeit der Kündigungsregel von wann bis wann CRMD_CANCEL_IR:: CANCVALFROM_ID und CRMD_CANCEL_IR::CANCVALTO_ID.

PREISFINDUNGSPARAMETER-SET

Die CRMD_PRICING Tabelle stellt Informationen für die Preisfindung eines Geschäftsvorgangs, bzw. Positionen bereit. Nur wenn ein Geschäftsvorgang einen Verweis auf dieses Set besitzt, kann für eine Vorgangsposition eine Preisfindung durchgeführt werden.

Feldname	Beschreibung
TAX_DEST_CTY	Land
TAX_DEST_REG	Region
AC_INDICATOR	Berechnungsmotiv
CURRENCY	Währung
EXCHG_TYPE	Währungskurstyp
PMNTTRMS	Zahlungsbedingung
CUST_GROUP	Kundengruppe
PRICE_LIST	Preislistentyp

PRICE_GRP	Kundenpreisgruppe
ETAX_SOURCE	Verbrauchsteuersatzherkunft
ETAX_HAND_TYPE	Verbrauchsteuerabfertigungsart

Tabelle 15: Felder der Tabelle CRMD_PRICING

Zu den preisfindungsrelevanten Informationen gehört die Währung, die Zahlungsbedingung, die Kundengruppe, die Kundenpreisgruppe, der Preislistentyp, die Verbrauchsteuerabfertigungsart, die Verbrauchsteuersatzherkunft und das Berechnungsmotiv.

Das Land und die Region sind für Steuerberechnungen wichtig. Je mehr Felder, sprich je mehr Preisfindunsgparameter angegeben sind, desto detaillierter und spezialisierter ist die Preisfindung. Z.B. kann es branchenspezifische Preis- und Steuerberechnungen geben.

Folgende Geschäftsvorgangsarten können ein Preisfindungsparameter Set besitzen: Fakturaanforderung, Finanzierungsvertrag, Lizenzeinkaufsvertrag, Lizenzverkaufsvertrag, Opportunity, Produktänderungsanweisung, Servicerückmeldung, Servicevertrag, Servicevorgang, Verkaufskontrakt und Verkaufsvorgang

QUALIFIKATIONSANFORDERUNG-SET

CRMD_QUALIF enthält Felder mit den Qualifikationsanforderungen an Servicemitarbeiter hinterlegt werden können, die Servicetätigkeiten ausführen.

Feldname	Beschreibung
OBJID	Objekt-ID der Qualifikation
VALIDFROM	Gültigkeitsbeginn
VALIDTO	Gültigkeitsende
OBLIGATORY	Anforderung obligatorisch?
PRFCY_MIN	Qualifikationsausprägung Minimum
PRFCY_MAX	Qualifikationsausprägung Maximum
PRFCY_OPT	Qualifikationsausprägung Optimum

Tabelle 16: Felder der Tabelle CRMD_QUALIF

Es kann genau angegeben werden, welches das minimale, maxiamale und optimale Qualifikationsprofil sein muss und von wann bis wann diese Qualifikationsanforderung gebraucht wird. Relevant ist das Qualifikationsanforderungs Set für die Geschäftsvorgänge Reklamtion und Servicevorgang.

STEUER-SET

BBP_PDTAX bildet Daten rund um die Umsatzsteuer eines Geschäftsvorgangs ab. Z.B. kann dieses Set dazu verwendet werden, die Umsatzsteuer einer Lieferantenrechnung in ein Buchungssystem zu überführen.

Feldname	Beschreibung
TAX_CODE	Umsatzsteuerkennzeichen
TAX_AMOUNT	Steuerbetrag
TAX_CODE_ORIGIN	Herkunft Steuerkennzeichen
TAX_JURCODE_TO	Tax Jurisdiction Code - Standort für

	Steuerrechnung
TAX_JCD_TO_ORG	Herkunft des Tax Jurisdiction Code
CURRENCY	Währungsschlüssel
TAX_BASE_AMOUNT	Basiswert zur Steuerermittlung

Tabelle 17: Felder der Tabelle BBP_PDTAX

Einzig dem Geschäftsvorgang Lieferantenrechnung kann dieses Set angefügt werden.

SURVERYBEWERTUNG-SET

Unter einem Survey im Kontext von Geschäftsvorgängen werden Fragebögen, Bewertungsbögen und Umfragen für Geschäftsvorgänge verstanden. Diese Dokumente werden dem Geschäftsvorgang oder dessen Positionen über das Surveybewertungs Set zugeordnet.

Die Inhalte und der Aufbau der Formulare erfolgt über die sogenannte CRM Survey Suite. Je nach Typ von Geschäftsvorgang werden bei dessen Bearbeitung die Surveyformulare ausgefüllt und über das Set im Hintergrund mit dem Vorgang verbunden.

Die Survey Formulartypfindung für einen Geschäftsvorgang ist Sache des Customizing und wird daher noch eingehender betrachtet.

VERKAUF-SET

Die Attribute der CRMD_SALES Tabelle beinhalten verkaufsspezifische Daten zu einem Geschäftsvorgang. Es gibt eine ganze Reihe von Geschäftsvorgangsarten oder deren Positionen, die selbst für einen Verkaufsvorgang relevant werden können. Dazu gehören: Fakturaanforderung, Lizenzeinkaufsvertrag, Lizenzverkaufsvertrag, Opportunity, Partnerdesignprojekt, Produktänderungsanweisung, Reklamation, Servicerückmeldung, Servicevertrag, Servicevorgang, Verkaufskontrakt und Verkaufsvorgang. Die folgende Tabelle zeigt die wichtigen Felder von CRMD_SALES, welches die verkaufsspezifischen Informationen sind.

Feldname	Beschreibung
REGION	Region
INDUSTRY	Branche
PCAT_HDR_GUID	GUID eines Produktkataloges (Kopf)
PCAT_VRT_GUID	GUID einer Produktkatalogvariante
PCAT_CTY_GUID	GUID eines Produktkatalogbereiches
CUST_GROUP1	Kundengruppe 1 (noch vier weitere vorhanden)
PO_NUMBER_SOLD	Externe Referenznummer des Auftraggebers
PO_DATE_SOLD	Datum des Referenzbelegs
YOUR_REF_SOLD	Zeichen des Auftraggebers
PO_NUMBER_SHIP	Externe Referenznummer des Warenempfängers
PO_DATE_SHIP	Datum des Referenzbelegs Warenempfänger
YOUR_REF_SHIP	Zeichen des Warenempfängers
USAGE_TYPE	Verwendungskennzeichen
REJECTION	Absagegrund
PROBABILITY	Wahrscheinlichkeit des Auftrags
DATE_QTY_FIXED	Datum und Menge
REQ_DLV_DATE	Wunschlieferdatum

REQ_TIMEZONE	Zeitzone Wunschlieferdatum
RULE_REQ_DATE	Regel Wunschlieferdatum

Tabelle 18: Felder der Tabelle CRMD_SALES

Die externen Referenznummern PO_NUMBER_SOLD des Auftraggerbers und PO_NUMBER_SHIP des Auftragnehmers sind für finanzielle Transaktionen, sprich für finale Buchungen in einem Buchungssystem Voraussetzung, da sonst keine Buchungen erfolgen können.

VERSAND-SET

Analog zum Verkauf sind im Versand Set versandspezifische Informationen in der Tabelle CRMD_SHIPPING hinterlegt.

Feldname	Beschreibung
INCOTERMS1	Incoterms Teil 1
INCOTERMS2	Incoterms Teil 2
SHIP_COND	Versandbedingung
DELI_UNLIM_TOL	Unbegrenzte Überlieferung erlaubt - ja/nein
OVER_DLV_TOL	Toleranzgrenze für Überlieferung
UNDER_DLV_TOL	Toleranzgrenze für Unterlieferung
DLV_PRIO	Lieferpriorität
PART_DLV	Liefersteuerung
DLV_GROUP	Liefergruppe (Positionen werden zusammen ausgeliefert)
DELIVERY_BLOCK	Evtl. Liefersperrgrund des Geschäftspartners
PART_DLV_ITM	Liefersteuerung der Position
TRANS_MOT	Verkehrszweig

Tabelle 19: Felder der Tabelle CRMD_SHIPPING

Viele der Geschäftsvorgänge, die das Verkauf Set nutzen können auch auf das Versand Set referenzieren. Dies sind: Lizenzeinkaufsvertrag, Lizenzverkaufsvertrag, Reklamation, Service-rückmeldung, Servicevertrag, Servicevorgang, Verkaufskontrakt und Verkaufsvorgang.

WERTLIMIT-SET

Ein Wertlimit, sprich ein Datensatz aus der Tabelle BBP_PDLIM, legt eine maximale Grenze fest, wie viele Produkte ein entsprechender Geschäftsvorgangstyp beinhalten kann. Ein Anwendungsbeispiel: Über einen bestimmten Zeitraum werden von einem Geschäftsvorgangstyp Bestellung eine große Menge von Produkten abgerufen. Da aufgrund der Höhe dieser geforderten Menge keine einmalige Bestellung möglich ist, wird die Gesamtmenge auf mehrere Bestellungen gesplittet. Die nachstehende Tabelle listet die Attribute auf, die Werte beinhalten, die ein Wertlimit definieren.

Feldname	Beschreibung
LIMIT	Gesamtlimit
UNLIMITED	Unbegrenztes Limit – ja/nein
EXP_VALUE	Erwarteter Wert
CURRENCY	Währungsschlüssel
CATEGORY	Produktkategorie – ID
FINAL_INV	Endrechnungskennzeichen
FINAL_ENTRY	Enderfassungs-Kennzeichen (Waren/Leistungen)
DEL_IND	Löschkennzeichen Einkaufsbeleg
LIM_REF_H_ID	Kopfnummer des Objektes, auf das sich das Limit bezieht
LIM_REF_I_ID	Positionsnummer des Objektes, auf das sich das Limit bezieht
VAL_CF_E	Finanzieller Wert der erfassten Rückmeldung
VAL_CF	Finanzieller Wert der freigegebenen Rück-meldung
VAL_IV_E	Rechnungswert erfaßt
VAL_IV	Rechnungswert freigegeben
NUM_CONF	Anzahl erfaßter Bestätigungen zu einer Bestellung
NUM_INV	Anzahl erfaßter Rechnungen einer Bestellung oder Bestätigung

Tabelle 20: Felder der Tabelle BBP_PDLIM

TERMIN-SET

Ein Termin der sich aus Daten der Tabelle SCAPPTSEG zusammensetzt, kann auf drei Arten definiert werden: 1. als Zeitpunkt, 2. als Zeitintervall, begrenzt durch Anfangs- und Endzeitpunkt und 3. als Zeitdauer von Endzeitpunkt minus Anfangszeitpunkt.

Ferner bestimmt die Terminart die Bedeutung eines Termins. Je nach Anwendungsart, beispielsweise Verträge oder Geschäftsvorgänge wie Aktionsarten legen jeweils eine andere Bedeutung eines Termins fest. Ein Termin in einem Vertrag kann bei dessen Verfehlung finanzielle oder gar gerichtiliche Folgen haben. Bei einem Geschäftsvorgang ist eine sich anbahnende Terminverfehlung evtl. durch verschiedene Maßnahmen abwendbar und bleibt dann ohne Folgen.

Termine können auch strukturiert werden. So gibt es Terminprofile, die Terminarten zusammenfassen. Für eine Terminart kann eine Terminregel für die Generierung von Vorschlags-werten defininiert werden.

Feldname	Beschreibung
APPT_GUID	ID des Termins
TST_FROM	Zeitstempel des Anfangszeitpunkts des Termins
TST_TO	Zeitstempel des Endezeitpunkts des Termins
ZONE_FROM	Zeitzone des Anfangszeitpunktes des Termins
ZONE_TO	Zeitzone des Endezeitpunktes des Termins
APPT_TYPE	Technischer Name der Terminart
ENTRY_BY	Erfasser
ENTRY_TST	Zeitstempel des Anlegezeitpunkts
CHANGE_BY	Letzter Änderer
CHANGE_TST	Zeitstempel des Änderungszeitpunkts

Tabelle 21: Felder der Tabelle SCAPPTSEG

Es gibt nur zwei Geschäftsvorgänge, die einen Verweis auf ein Termin Set haben können: Einkaufskontrakt und den Einkaufslieferplan. Auf Positionsseite gibt es sehr viele Positionstypen, denen ein Termin zugeordnet werden kann. Einige von ihnen schreiben sogar vor einen Verweis auf einen Termin Set Datensatz vor.

3.4.3 Customizing Tabellen des Geschäftsvorgangs und dessen Positionen

Das Customizing des Geschäftsvorgangs lässt sich in drei Teile gliedern:

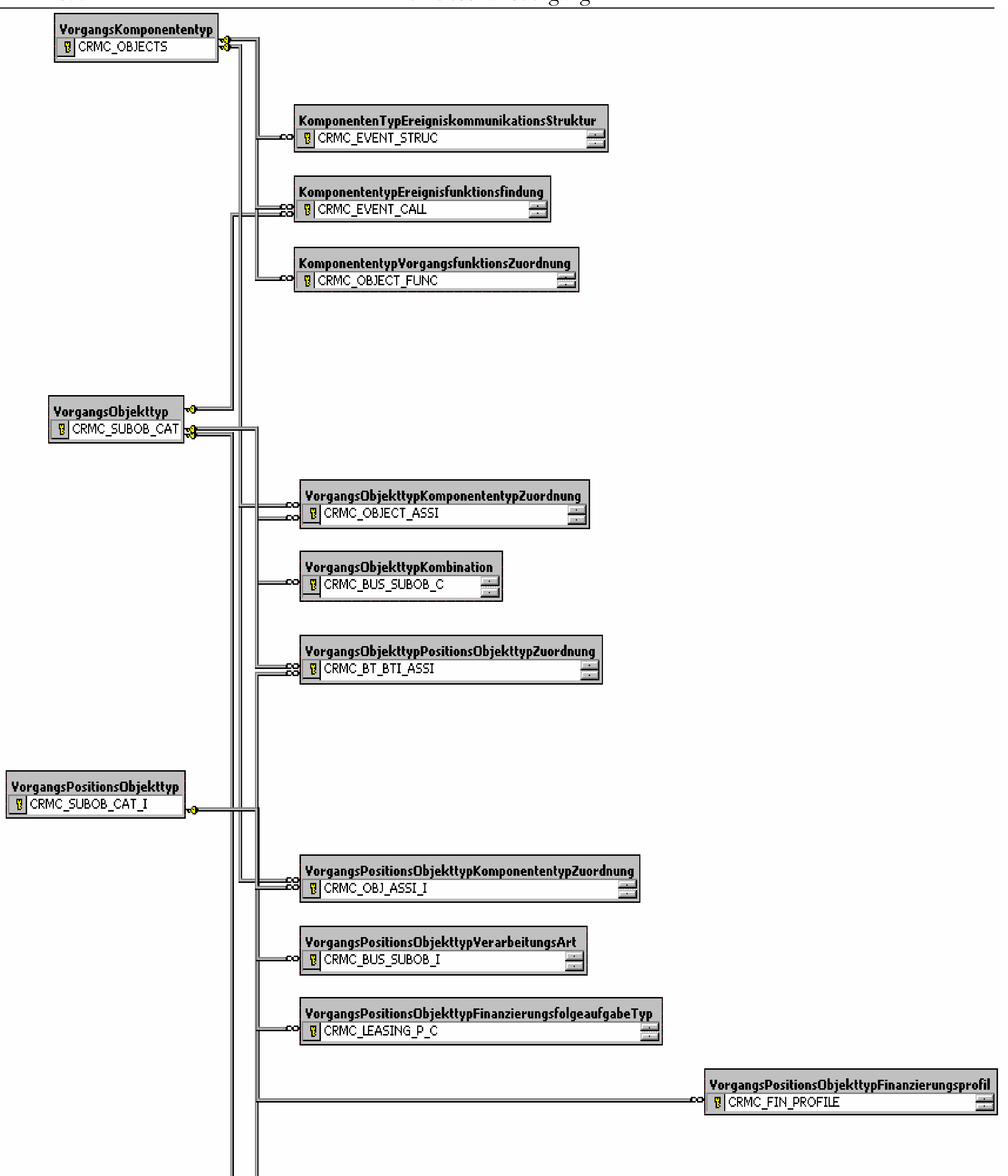
1. Struktur des Geschäftsvorgangs
2. Verarbeitung von Geschäftsvorgängen
3. Verarbeitung von Geschäftsvorgangspositionen

Die **Strukturierung** von des Geschäftsvorgängen –und positionen erfolgt durch GESCHÄFTS-VORGANGSKOMPONENTENTYPEN, bzw. VORGANGSPOSIITONSKOMPONENTENTYPEN und GESCHÄFTSVORGANGSOBJEKTTYPEN, bzw. VORGANGSPOSITIONSOBJEKTTYPEN.

Komponententypen sind Informationseinheiten nach betriebswirtschaftlichen Kriterien. Vorgangsobjekttypen definieren die existierenden betriebswirtschaftlichen Vorgänge. Vorgangs-positionsobjekttypen legen fest, welche Positionstypen für einen Geschäftsvorgang erlaubt sind.

Die Vorgangsart, also die **Verarbeitung von Geschäftsvorgängen**, bestimmt für einen Vorgangsobjekttyp Vorgabewerte für den angegebenen Vorgangsobjekttyp. Die Vorgangs-positionsart, sprich die **Verarbeitung von Geschäftsvorgangspositionen** definiert für einen Vorgangspositionsobjekttyp analog die Verarbeitung von Geschäftsvorgangspositionen für den angegebenen Vorgangsobjekttyp. Beispiele für Vorgangsart sind ein Kontakt, eine Aufgabe, ein Angebot oder ein Einkaufslieferplan.

Das folgende Schaubild zeigt das Entity Relationship Modell, ERM für das Customizing des Geschäftsvorgangs.


VorgangsKomponententyp
CRMC_OBJECTS
KomponentenTypEreigniskommunikationsStruktur
CRMC_EVENT_STRUC
KomponententypEreignisfunktionsfindung
CRMC_EVENT_CALL
KomponententypVorgangsfunktionsZuordnung
CRMC_OBJECT_FUNC
VorgangsObjekttyp
CRMC_SUBOB_CAT
VorgangsObjekttypKomponententypZuordnung
CRMC_OBJECT_ASSI
VorgangsObjekttypKombination
CRMC_BUS_SUBOB_C
VorgangsObjekttypPositionsObjekttypZuordnung
CRMC_BT_BTI_ASSI
VorgangsPositionsObjekttyp
CRMC_SUBOB_CAT_I
VorgangsPositionsObjekttypKomponententypZuordnung
CRMC_OBJ_ASSI_I
VorgangsPositionsObjekttypVerarbeitungsArt
CRMC_BUS_SUBOB_I
VorgangsPositionsObjekttypFinanzierungsfolgeaufgabeTyp
CRMC_LEASING_P_C
VorgangsPositionsObjekttypFinanzierungsprofil
CRMC_FIN_PROFILE

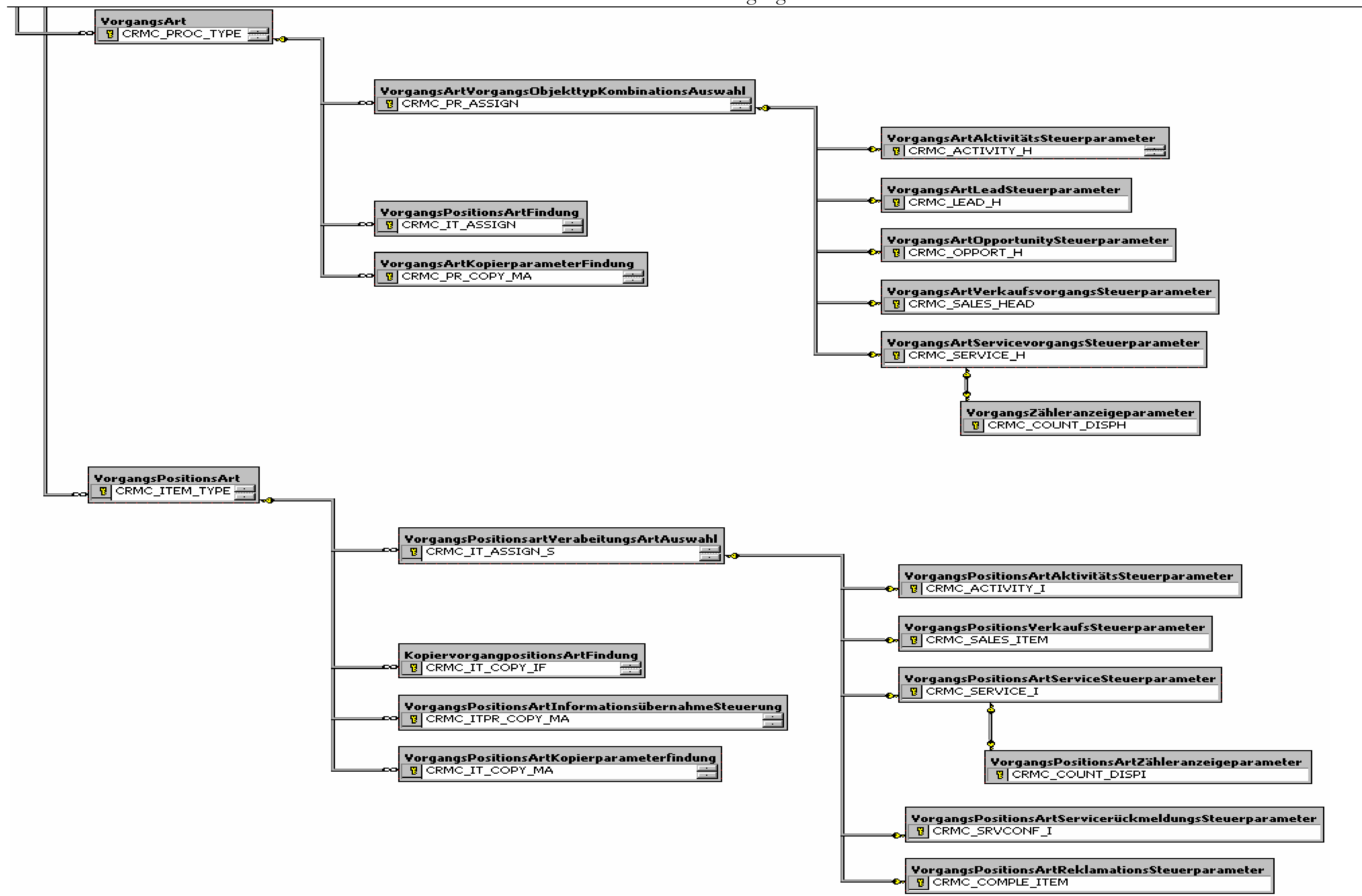


Abbildung 23: Das Entity Relationship Modell der Customizing Tabellen des Geschäftsvorgangs

Es gilt zu klären, wie die Struktur des Customizing von Geschäftsvorgängen datenmodell-technisch realisiert ist, und wie die Zusammenhänge von Vorgangskomponententypen, Vorgangsobjekttypen und Vorgangsarten, sowie den Positionen sind. Um dies zu tun, wird folgender-maßen vorgegangen: Es werden zuerst die Tabellen aufgezählt die im Datenmodell miteinander verknüpft sind. Anschließend wird geklärt, wie, d.h. über welche Tabellenfelder, sie verbunden sind. Im letzten Schritt erfolgt die Klärung der betriebswirtschaftlichen Bedeutung der verknüpf-ten Entitäten in ihrem Zusammenspiel. Dort wo es angebracht ist, wird auch noch auf die Sicht des Entwicklers eingegangen, d.h. welche Funktionsbausteine bei der Realiserung der betriebs-wirtschaftlichen Bedeutung von Belang sind.

DER GESCHÄFTSVORGANGSKOMPONENTENTYP

Im Datenmodell wird der der Geschäftsvorgangskomponententyp durch die Tabelle CRMC_ OBJECTS repräsentiert. Zum Vorgangskomponententyp gehören:

- Die Ereigniskommunikationsstruktur CRMC_EVENT_STRUC
- Die Ereignisfunktionsfindung CRMC_EVENT_CALL
- Die Zuordnungstabelle von Funktionen zum Komponententyp CRMC_OBJECT_FUNC

Realisiert werden wird die Verknüpfung von CRMC_OBJECTS zu diesen drei Tabellen über eine 1:n Beziehung über das Schlüsselattribut CRMC_OBJECTS::NAME. Einem Geschäftsvorgangskomponententyp können mehrere Ereigniskommunikationen, mehrere Ereignisfunktionsfindungen und mehrere Vorgangsfunktionszuordnungen zugewiesen sein. Verknüpft wird CRMC_EVENT_STRUC mit CRMC_EVENT_STRUC::OBJ_NAME und CRMC_OBJECTS::NAME, CRMC_EVENT_CALL mit CRMC_EVENT_CALL::OBJ_NAME und CRMC_OBJECTS::NAME und CRMC_OBJECT_FUNC mit CRMC_OBJECT_FUNC:: NAME und CRMC_OBJECTS::NAME.

Bei Geschäftsvorgängen gibt es Ereignisse. Z.B. nach der Erstellung, vor einer Speicherung, eine Bestätigung, nach dem Löschen eines Geschäftsvorgangs und andere. Welches dieser Ereignisse eingetreten ist, bildet das Attribut CRMC_EVENT_STRUC::EVENT ab. Ein Ereignis aus der Ereigniskommunikationsstruktur ist überdies einer Ereigniskategorie zugeordnet. Dies wird über die Verbindung von CRMC_EVENT_STRUC::EVENT und CRMC_EVENTS::EVENT erreicht. Die Tabelle CRMC_EVENTS besitzt ein Feld CRMC_EVENTS::CATEGORY, das Werte für die Ereigniskategorien Initialisieren, Prüfen, Ändern, Löschen und Sichern enthält.

Benötigt werden Ereignisse für die Publizierung von Änderungen von Geschäftsvorgängen an andere Vorgangskomponenten, wie die Set Tabellen. Ein Beispiel für die Set Tabelle CRMD_ PRICING: Ändert sich für einen Geschäftsvorgang, der einen Link auf Preisfindungsparameter Set hat, das Berechnungsmotiv CRMD_PRICING::AC_INDICATOR, von 100 prozentiger Kulanz auf ein allgemeines Berechnungsmotiv, muss ein Ereignis durch den Event Handler Service an betroffene Set Komponenten kommuniziert werden. Es kann beispielsweise möglich sein, dass durch die Änderung des Berechnungsmotivs, eine Neuberechnung des Preises erforderlich ist. In diesem Fall muss der betroffene Geschäftsvorgang betriebswirtschaftlich konsistent bleiben. Dies funktioniert aber nur dann, wenn durch die Benachrichtung der Preiskalkulations Set Komponente entsprechende Anpassungen und Änderungen vorgenommen werden.

Wie der Event Handler Ereignisse, die von Änderungen in Komponenten ausgelöst wurden, an betroffene andere Komponenten weiterleitet, verdeutlicht das folgende Schaubild.

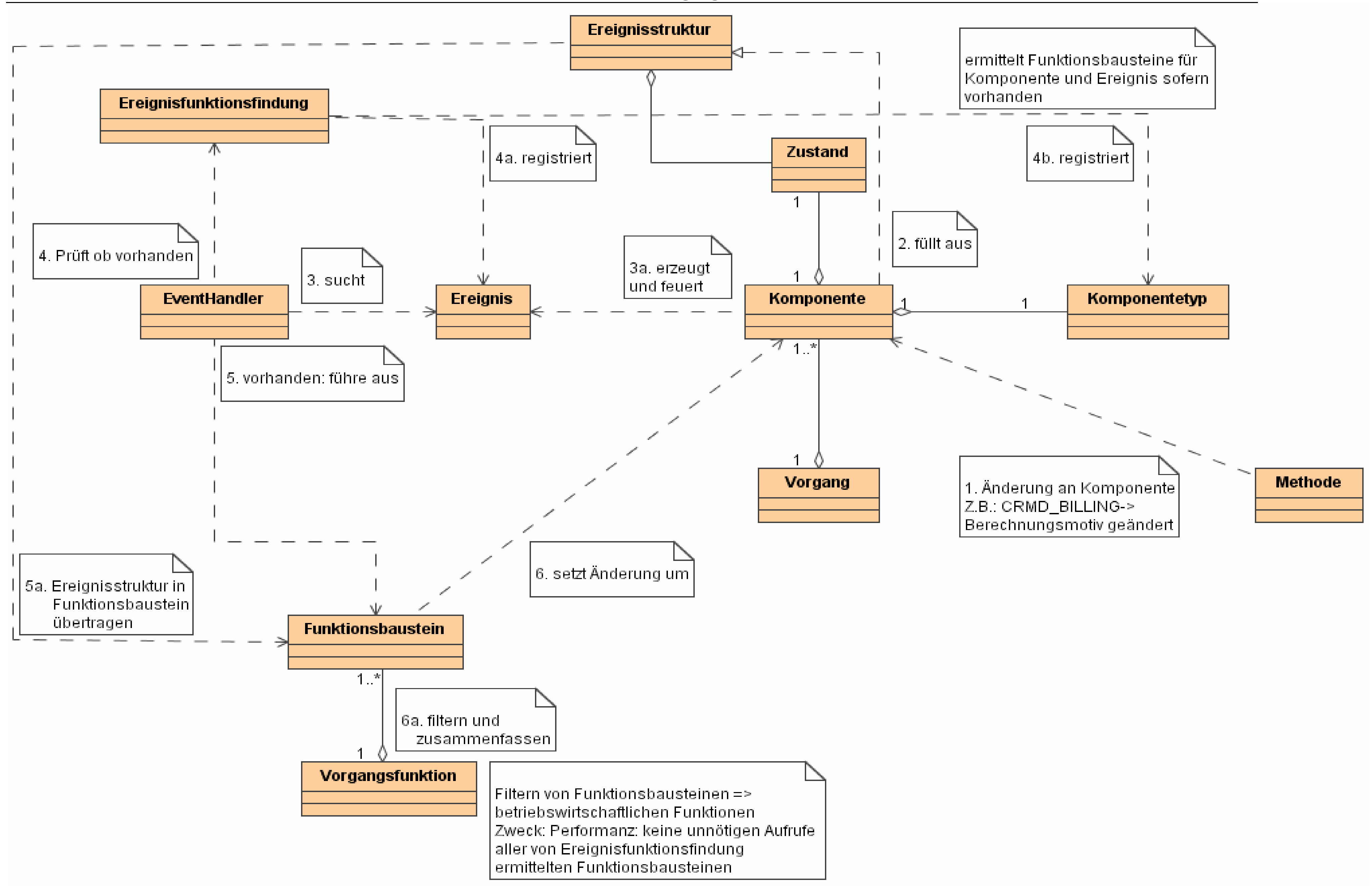


Abbildung 24: Darstellung der Weiterleitung von Ereignissen durch Änderungen an Komponenten an andere Komponenten durch den Event Handler

DER GESCHÄFTSVORGANGS(POSISIONS)OBJEKTTYP

Geschäftsvorgangsobjekttypen sind Klassifizierung der Geschäftsvorgänge in Subtypen zuständig. Die Tabelle im Datenmodell CRMD_SUBOB_CAT besitzt hierfür lediglich zwei Felder: CRMC_SUBOB_CAT::SUBOBJ_CATEGORY und CRMD_SUBOB_CAT::APPLICATION. Das Primärschlüsselfeld CRMC_SUBOB_CAT::SUBOBJ_CATEGORY ist die Verbinding zur Geschäftsvorgangsgrunddatentabelle CRMD_ORDERADM_H. Über CRMD_ORDERADM_H::OBJECT_TYPE sind die beiden Entitäten miteinander verknüpft, sodass jeder Geschäftsvorgang ungeachtet seiner Struktur immer genau einen Geschäftsvorgangstyp besitzt.

Die datentechnische Verbindung zwischen Geschäftsvorgangsobjekttyp und Geschäftsvorgangskomponententyp wird über die n:m Verknüpfungstabelle CRMC_OBJECT_ASSI verwirklicht. Sie besitzt zwei Verknüpfungsattributfelder, CRMC_OBJECT_ASSI::SUBOBJ_CATEGORY zur Identifizierung des Geschäftsvorgangsobjekttyp und CRMC_OBJECT_ASSI::NAME für den zugehörigen Komponententyp.

Darüber hinaus können einerm Geschäftsvorgangsobjekttyp mehrere Geschäftsvorgangspositionsobjekttypen zugewiesen werden, um festzulegen, welche Positionen zu einem Geschäftsvorgang gehören können. In diesem Fall fungiert der Geschäftsvorgangsobjekttyp als führender Objekttyp des jeweiligen Geschäftsvorgangs. Die Verbindung zwischen dem Geschäftsvorgangsobjekttyp CRMC_SUBOB_CAT und dem Geschäftsvorgangspositionsobjekttyp CRMC_OBJ_CAT_I funktioniert über die Vorgangsobjekttypkombinationen. Eine Vorgangsobjekttypkombination ist ein Datensatz aus der Tabelle CRMC_BUS_SUBOB_C. Diese besitzt das Schlüsselfeld CRMC_BUS_SUBOB_C::OJ_NAME_GENERAL, das den übergeordneten Geschäftsvorgangsobjekttyp repräsentiert. Die Objekttypkombinationen werden durch die Attribute CRMC_BUS_SUBOB_C::CATEGORY_1 bis CRMC_BUS_SUBOB_C::CATEGORY_15 abgebildet. Jedem dieser CATEGORY_X Felder kann ein Geschäftsvorgangspositionsobjekttyp zugewiesen werden, sodass dadurch die maximale Struktur eines Geschäftsvorgangs festgelegt ist.

DIE GESCHÄFTSVORGANGS(POSITIONS)ART

Ein Vorgangsart oder Vorgangspositionsart lässt sich mit Verarbeitungsregel oder mit Steuerungsparameter beschreiben. Diese Regeln werden ebenfalls durch Tabellen abgebildet, siehe Abbildung 23. Für eine Vorgangsart oder Vorgangspositionsart, es ist mit Vorgangsart im Folgenden beides gemeint, muss ein Vorgangsobjekttyp, bzw. Vorgangspositionsobjekttyp existieren, dem eine Vorgangsart, bzw. Vorgangspositionsart zugeordnet werden kann.

Für ein Lead beispielsweise, also einem Geschäftsvorgang mit dem Vorgangsobjekttyp CRMD_ORDERADM_H::OBJECT_TYPE Lead gibt es in der Steuerungsparametertabelle CRMC_LEAD_H ein Attribut CRMC_LEAD_H::LEAD_TYPE, für das verschiedene Leadtypen definiert werden können, z.B. Schulung oder Beratung. Auch für den Geschäftsvorgang Opportunity gibt es solche Steuerungsparameter. In der Tabelle CRMC_OPPORT_H kommen u.a. folgende Steuerungsattribute vor:

- CRMC_OPPORT_H::SALESCYCLE: Verkaufszyklus, z.B. Neukundengeschäft
- CRMC_OPPORT_H::TYPE: Opportunitygruppe wie Neukunden oder Bestandskunden
- CRMC_OPPORT_H::OPPDRG_FLAG: selbstdefinierte Opportunityarten
- CRMC_OPPORT_H::TEXT_PROC_COMP: selbstdefinierte Textschemata für Wettbewerber

Die Verbindung von Vorgangsart und Vorgangsobjekttyp besteht in der 1:n Beziehung zwischen den Entitätsschlüsselattributen CRMC_SUBOB_CAT::SUBOBJ_CATEGORY und CRMC_PROC_TYPE::OBJECT_TYPE. Einem Vorgangsobjekttyp können also mehrere Vorgangsarten zugeordnet sein. Analog ist diese Beziehung zwischen Vorgangspositionsart und Vorgangs-

positionsobjekttyp: CRMC_SUBOB_CAT_I::SUBOBJ_CATEGORY und CRMC_ITEM_TYPE::OBJECT_TYPE.

Auch die Zuordnung von Steuerungsparametertabellen zu Vorgangsarten ist über eine Zwischentabelle CRMC_PR_ASSIGN geregelt. Auf Seiten der Positionsarten ist die Tabelle CRMC_IT_ASSIGN das Bindeglied zwischen den Vorgangspositionsarten CRMC_ITEM_TYPE und den Steuerparametertabellen. CRMC_PR_ASSIGN und CRMC_IT_ASSIGN werden auch Kombinationsauswahlen genannt. Sie bestimmen, ob einer Vorgangsart CRMC_PROC_TYPE Steuerungsparameter zugewiesen werden können oder nicht. Steuerungsparameter können einer Vorgangsart zugeordnet werden, wenn in der Tabelle CRMC_PR_ASSIGN für eine Vorgangsart CRMC_PROC_TYPE::PROCESS_TYPE in der Tabelle CRMC_PR_ASSIGN für CRMC_PR_ASSIGN::PROCESS_TYPE Werte für das Attribut CRMC_PR_ASSIGN::SUBOBJ_CATEGORY vorliegen.

Z.B. gibt es für die Vorgangsart Lead eine Zuweisung für die Lead Steuerparametertabelle CRMC_LEAD_H, weil in der Vorgangsartobjekttypkombinationsauswahltabelle CRMC_PR_ASSIGN für den CRMC_PR_ASSIGN::PROCESS_TYPE Wert Lead der Wert BUS2000108 des Attributs CRMC_PR_ASSIGN::SUBOBJ_CATEGORY existiert. BUS2000108 steht für den Vorgangsobjekttyp Lead.

Nun stellt sich noch die Frage, wie geregelt ist, welche Vorgangspositionsarten einer Vorgangsart zugewiesen sein können. Die für diese Zuordnung entscheidenden Felder der Tabelle CRMC_IT_ASSIGN sind:

- CRMC_IT_ASSIGN::PROCESS_TYPE für die Geschäftsvorgangsart
- CRMC_IT_ASSIGN::ITEM_TYPE_MAIN für den Positionstyp der übergeordnete Position sofern vorhanden
- CRMC_IT_ASSIGN::ITM_TYPE bis ITM_TYPE_ALTX geben mögliche Positionstypen für die spezifische Positionsart an. Das X bei CRMC_IT_ASSIGN::ITM_TYPE_ALTX steht für eine laufende Nummer von eins bis elf.

3.5 Zusammenfassung

Der Geschäftspartner und der Geschäftsvorgang sind nebem dem Produkt und der Preis- und Konditionstechnik die wichtigesten Bausteine des SAP CRM 4.0. Betriebswirtschaftlich sind Geschäftspartner und Geschäftsvorgänge eng miteinander verzahnt. Wenn es um das Management von Kundenbeziehungen geht, sind Geschäftsvorgänge immer einem Kunden, also einem Partner zuzuordnen, da für den Kunden Produkte hergestellt, oder Dienstleistungen erbracht werden, die mit Geschäftsvorgängen in Beziehung stehen.

Um in SAP CRM 4.0 alle möglichen Anwendungsfälle von Unternehemen und deren Strukturen und Prozesse abdecken zu können, sind die Datenstrukturen von Geschäftspartner und Geschäftsvorgang recht komplex. Bei näherem Hinsehen ist aber festzustellen, dass es eine strikte und übersichtliche Ordnung gibt, mit der sich verschiedene Tabellentypen für unterschiedliche Funktionen und Eigenschaften unterscheiden lassen.

Für Sachverhalte, die für alle Anwendungsfälle gelten, gibt es die Basistabelle BUT000 für den Geschäftspartner, CRMD_ORDERADM_H und CRMD_ORDERADM_I sind die Grundtabellen Geschäftsvorgang und Geschäftsvorgangsposition für den Geschäftsvorgang. Extension- und Set Tabellen beinhalten ausgelagerte zusaätzliche Informationen. Customizing Tabellen dienen zur Abibldung individueller Sturkturen und Eigenschaften einer konkreten SAP CRM Installation für ein Unternehmen.

4 SAP Web Dynpro

Bisher wurden für Webanwendungen unter SAP die BSPs, die Business Server Pages verwendet. Der Name BSP ist an JSP, bzw. ASP angelehnt und so werden bei den BSPs HTML und ABAP Code direkt oder über Tag Libraries referenziert, zusammen vermengt. Das Ergebnis ist eine Webbenutzeroberfläche, mit der auf Daten eines SAP R3 oder SAP CRM Systems zugegriffen werden kann.

Mit Web Dynpro hat SAP einen Webframework entwickelt, das wie Struts und JSF eine strikte Trennung von Daten, Darestellung und Anwendungssteuerung vornimmt. Wesentlich konsequenter als dies bei den BSPs der Fall ist. Zwei weitere fundamentale Unterschiede zu den BSP ist zum einen die Verbindung von Java und ABAP anstatt ABAP und HTML. Moderner heutiger Softwareentwicklungsstandards entsprechend wird bei Web Dynpro die modelelierungsgetriebene Anwendnungsentwicklung unterstützt. Das Viewdesign und Navigationsverhalten beispielsweise wird komplett modelliert, anstatt es direkt in den BSP oder HTML Code zu schreiben.

Auch findet sich in Darstellungskomponenten keinerlei ABAP Code. ABAP Funktionsbausteine sind komplett von jeglichen Anwendungsbestandteilen getrennt und werden lediglich von Steuerungskomponenten angestoßen und deren Ergbnisse zur weiteren Verwendung aufgefangen.

Ein wichtiger Unterschied des Webentwicklungsframeworks Web Dynpro ist im Vergleich zu Struts und JSF der, dass dieser nicht unter einer Open Source Lizenz frei zur Verfügung steht. Er ist Teil der Auslieferung des SAP Web Application Servers und benötigt diesen auch für den Betrieb. Aufgrund der fehlenden Open Source Lizenz steht der Quelltext von Web Dynpro nicht für eine detaillierte Quelltextanalyse zur Verfügung wie bei den Kapiteln von Struts und JSF.

4.1 Kapitelübersicht

Um dem Leser dieses Framework bekannt zu machen, wird folgendermaßen vorgegangen: im ersten Teil dieses Kapitels, 4.2, wird eine praktische Programmieranleitung gegeben, welche Aktionen der Leser, bzw. Entwickler durchführen muss, um mittels eines Web Dynpro Projekts im SAP Netweaver Developer Studio eine Web Dynpro Webapplikation zu erstellen.

Die Webentwicklung mit Web Dynpro ist momentan sehr modern, so modern, dass sie noch kaum in der praktischen Anwendung zu finden ist. Bisher bedeutet Webanwendungsentwicklung mit SAP, das Entwicklen von Busniess Server Pages, den BSPs. Diesem Thema in Verbindung mit dem Web Application Server widmet sich [6]. Dem Java Connector Jco ist [5] gewidmet. Hierbei geht es nicht um Webapplikationen, sondern allgemeiner um den Zugriff auf ein SAP System aus Java heraus mithilfe der Middlewareschicht JCo. Der JCo ist jedoch ein fundamen-

taler Bestandteil für Web Dynpro, da Web Dynpro Applikationen, die auf dem Web Application Server laufen, durch dieses Framework den JCo nutzen, um mit einem SAP R/3 oder SAP CRM System kommunizieren. Der Programmierer muss bei der Praxis der Web Dynproanwendungsentwicklung jedoch nichts über den JCo wissen.

Im zweiten Teil, mit den Abschnitten 4.3 bis 4.7, wird der Aufbau des Web Dynpro Frameworks betrachtet, damit der der Programmierer Einsicht in die Arbeitsweise von Web Dynpro erhhält. Von zentraler Bedeutung ist hierbei die strenge Trennung zwischen Model- , Controller- und Darstel-lungskomponenten, wie dies auch bei Struts und JSF der Fall ist. Realisiert wird das MVC Muster bei Web Dynpro jedoch völlig anders. Kern dieser MVC Realisierung ist ein Javacodegenerator, der je nach Struktur und Aufbau der ABAP Funktionsbausteine, die die Modelkomponente repräsentieren, dem Design der Benutzeroberflächen und für die Controllerlogik, Javaklassen generiert, mit denen der Programmierer arbeitet.

Den generierten Klassen liegt das Web Dynpro Framework zugrunde, da die generierten Javaklassen aller drei MVC Komponenten auf Basisfunktionalitäten zurückgreifen, die in Web Dynpro Basisklassen und Interfaces entweder realisiert sind, wenn dies für jede Ausprägung der drei Komponenten unabhängig von deren individueller Struktur gleich ist. Oder deklariert, sprich speziell für eine MVC Komponente zu implementieren, wenn auf dessen spätere Struktur oder Aussehen Rücksicht genommen werden muss.

Die Javaarchitektur von Framework und generierten Klassen, sprich der Aufbau, die Verbindung und das Zusammenspiel von Model, View und Controller ist der rote Faden dieses Kapitels, nach dem sich auch der Aufbau des Kapitels vier orientiert.

4.2 Design und Konzeption einer SAP Web Dynpro Referenzanwendung

Um die Möglichkeiten des Web Dynpro Webentwicklungsframeworks praktisch zu erproben und zu lernen, sowie die Verwendung von Erkenntnissen aus dem Kapitel vier, verfolgt dieses Kapitel das Ziel, das Design und die Konzeption der SAP Web Dynpro Referenzanwendung zu dokumentieren. Dieses Dokumentationskapitel soll dem Entwickler, der noch nicht mit diesem Framework vertraut ist, in die Lage versetzen, sich eine solide Wissensbasis zur Entwicklung von Web Dynpro Applikationen zu erarbeiten.

Zwei Dinge sind hierbei von zentraler Bedeutung:

1. Der Umgang mit dem SAP Developer Studio für die modellier- und programmiertechnische Umsetzung der Konzeptions- und Designziele einer Web Dynpro Applikation.
2. Kenntnisse der zum Java Web Dynpro API gehörenden Bibliotheken mit den Funktionalitäten und den Aufgaben der jeweiligen Klassen. Stichwort Model – View – Controller.

In diesem Kapitel geht es schwerpunktmäßig um die eigentliche Praxis der Web Dynpro Entwicklung. Dazu gehört auch das Aufzeigen von Problemen und deren Lösungen, um dem Programmierer evtl. stundenlangen oder gar tagelangen Zeitverlust und Frust zu ersparen. Auf Problemquellen wird in den Unterkapiteln 4.2 gesondert hingewiesen. Punkt zwei ist das Thema der Kapitel 4.3 bis 4.7.

4.2.1 Features und Funktionalitäten der Referenzanwendung

Zu jeder Design- und Konzeptionsphase einer Applikationsentwicklung gehört u.a. vor allem die Ermittlung der gewünschten Funktionalitäten. Diese lauten wie folgt:

- Die Suche nach Geschäftspartnern nach einer Reihe optionaler Kriterien
- Die Auflistung relevanter Informationen zu einem Geschäftspartner, sprich nach der Suche eine Detailübersicht des selektierten Partners.
- Die Auflistung von verschiedenen Typen von Geschäftsvorgängen, die zu einem selektierten Geschäftspartner gehören.
- Die Anzeige von Detailinformationen zu einem selektierten Geschäftsvorgang.

4.2.2 Die Reihenfolge des Web Dynpro Entwicklungsprozesses

Nach dem Anlegen einer neuen Web Dynpro Anwendung im SAP Developer Studio wird durch die hierfür spezielle Web Dynpro Perspektive aller Quell- und aller Resourcedateien, sofort die strikte Trennung der Anwendungsschichten nach Model, View und Controller deutlich.

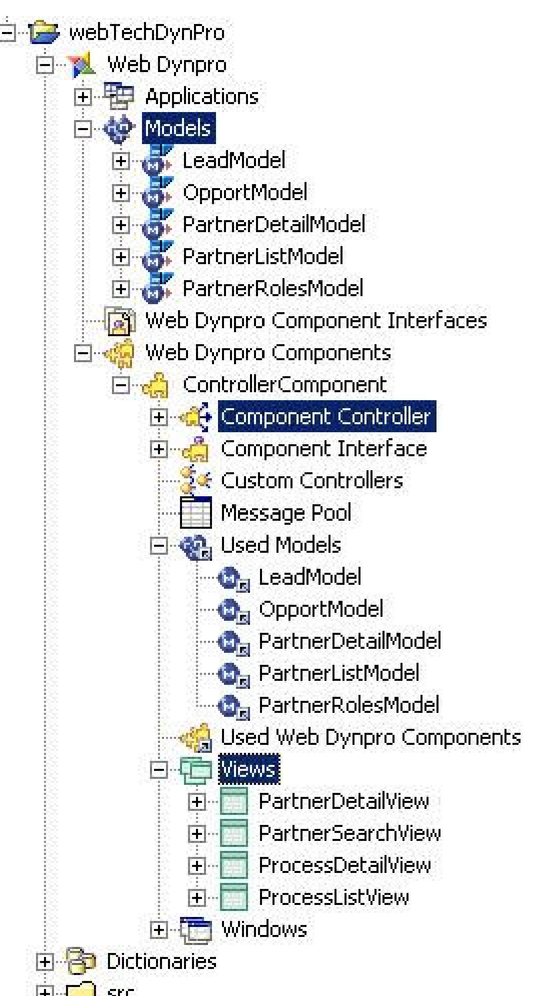


Abbildung 25: Ein Web Dynpro Projekt im Überblick in der Eclipse Perspektive des Web Dynpro Explorers

Die drei wichtigsten Unterknoten vom Knoten `Web Dynpro` sind diejenigen, die blau markiert sind. Unter `Models` sind alle Modelkomponenten eingeordnet die in der Applikation verwendet werden.

Die zweite Schicht ist die Controller Schicht. Alle Controller Komponenten einer Web Dynpro Anwendung sind unter dem Knoten `Web Dynpro Components` zu finden. Für kleinere Anwendungen wie die Referenzanwendung genügt eine Controller Komponente. Diese kann `Controller Component` heißen.

Um die Verbindung der Models zum Controller zu ermöglichen, sind unter dem Knoten `Used Models` alle Models nochmals aufgelistet, die auch dem Controller bekannt sind.

Die dritte Schicht der Views ist durch die Benutzeroberflächen unter dem Knoten `Views` repräsentiert, die die Daten der Modelkomponenten visualisieren, nachdem die Models vom Controller ausgelesen, oder bearbeitet wurden. Die Views sind direkt unter der erstellten `ControllerComponent` angeordnet.

Während der Implementierung der Funktionalitäten der Referenzanwendung hat sich eine bestimmte Herangehensweise als besonders fehlerunanfällig und praktikabel erwiesen. Begonnen wird mit dem Import der Modelkomponenten. Vorrausetzung ist, dass auf dem Ziel SAP System die entsprechenden Funktionsbausteine programmiert wurden und remote fähig sind. Die Remotefähigkeit ist notwendig, damit die Funktionsbausteine über das SAP Developer Studio auffindbar sind. Um einen Funktionsbaustein als Model in einem Web Dynpro Projekt verfügbar zu machen, sind folgende Schritte notwendig:

1. Rechtsklick auf den Knoten `Models` und Auswahl des Kontextmenupunktes `Create Model`
2. Auswahl von `Import Adaptive RFC Model`
3. Im folgenden Dialog sind folgende Daten einzugeben:
    a. `Model Name`: Gewünschter Name des Models unter dem Knoten `Models`. Für einen Funktionsbaustein, der z.B. Lead Daten ausliest kann der Name `LeadModel` sinnvoll sein.
    b. `Model Package`: Das Java Package wohin der SAP Developer Studio die generierten Javaklassen speichern soll, die für den Funktionsbaustein benötigt werden
    c. `Default logical system name for model instances` und `Default logical system name for RFC metadata`: Diese beiden Umgebungsvaribalen sind die Verknüpfung zwischen dem SAP Wep Application Server und dem SAP System. Die Einrichtung und das erfolgreiche Testen der Verbindung zwischen SAP System und dem SAP WAS sind Vorrausetzung für das Anlegen und Ausführen von Funktionsbausteinen/Models. Hierfür muss Konfigurationsaufwand auf Seiten des SAP Systems und des SAP WAS betrieben werden. Die genauen Schritte werden im Anhang C separat erläutert.
    d. `Logical Dictionary`: Wird automatisch mit dem ausgefüllt, was unter a. Eingetragen wird

Nach diesem Schritt müssen die Login Daten für das SAP System eingegeben werden. Im nächsten Fenster kann der Entwickler seine Auswahl des gewünschten Funktionsbausteins treffen. Darauf folgt die Generierung aller notwendigen Javaklassen, und das Model erscheint unter dem Knoten `Models` aus Abbildung 24. Die Javaklassen sowie andere Resource xml Dateien, die beim Codegenerierungsprozess für ein Model, bzw. ein Funktionsbautein erstellt werden, werden am Beispiel des Leadmodels und dessen Funktionsbaustein in Kapitel 4.2.3 erläutert.

Nachdem alle für die Web Dynpro Applikation nötigen Models integriert wurden, folgt nun die Erstellung einer Controller Komponente. Bei einem Rechtsklick auf den Knoten `Web Dynpro Components` und der Auswahl `Create Web Dynpro Component` im Kontextmenu wird nach Angabe von `Component Name` und `Component Package` im erscheinenden Dialog eine Controller Komponente erzeugt, sprich zum einen die für den Controller notwendigen Javaklassen und zum anderen der Controller Komponentenknoten unter `Web Dynpro Components` unter dem Namen, der im vorherigen Dialog unter `Component Name` eingetragen wurde.

Um die erstellten Modelkomponten dem gerade erstellten Controller bekannt zu machen, rechtsklickt man auf den Controller Komponentenunterknoten `Used Models` und wählt nach dem Klick auf `Add` des Kontextmenus im Dialogfenster diejenigen Models aus, mit denen der Controller verbunden sein soll. Damit wird erreicht, dass die generierte Controller Komponentenjavaimplementierungsklasse die Javaimplementierungsklassen der Modelkomponenten importieren und instanziieren kann.

Der für die Arbeit mit der Controller Komponente wichtigste Bestandteil ist der Controller Context. Dieser lässt sich als Datenspeicher beschreiben, in dem alle Datenstrukturen, sei es

diejenigen der Modelkomponenten, oder eigenständige, einfache Variablen für die programmiertechnische Verarbeitung und spätere Darstellung in den Benutzeroberflächen, hinterlegt sind. Genaueres zum Controller Context ist in Abschnitt 4.2.4 beschrieben.

Mit Ende dieses Arbeitsschrittes kommt der letzte Teil im Web Dynpro Entwicklungsprozess das Erstellen der Benutzeroberflächen und deren Verbindung zum Controller, sowie das Verbinden der Views untereinander, sprich dem Navigation Handling. Da diese beiden Punkte auf Wissen aus 4.2.4 aufbauen, erfolgt die Betrachtung dieses Themas in Kapitel 4.2.5.

Am Schluss des Entwicklungsprozesses kann die Anwendung gestartet, bzw. getestet werden. Dazu wird beim Hauptknoten `Applications` überhalb von `Models` über das Kontextmenu und `Create Application` ein Anwendungsstartobjekt erstellt. Unter Angabe von `Name` und `Package` im Dialogfenster kann die Anwendung über das Kontextmenu des gerade erschienenen Anwendungsstartobjekts die Application mittels `Deploy New Archive and Run` gestartet werden.

4.2.3 Die Ergebnisse der Modelcodegenerierung

Die Grundlage der Generierung aller Javaklassen und xml Resourcedateien für ein ABAP Model ist der Funktionsbaustein aus dem SAP System. Am Beispiel des Lead Models aus der Referenzanwendung wird gezeigt, welche Dateien anhand welcher Informationen wohin, d.h. in welche Verzeichnisse der Applikation abgelegt werden.

Der Funktionsbaustein für das Lead Model heisst Z_ALL_LEADS_BY_PARTNER_2. Seine Funktion ist es, alle Lead Daten aus den Tabellen CRMD_ORDERADM_H und CRMD_LEAD_H für einen Geschäftspartner auszulesen. Die Leaddaten für einen speziellen Geschäftspartner werden durch den Importparameter IM_PARTNER festgelegt, der vom Datentyp CRMD_ORDERADM_H::PARTNER ist, also dem Primärschlüssel dieser Tabelle. Die Suchergebnisse, sprich die Daten aus den Tabellen CRMD_ORDERADM_H und CRMD_LEAD_H werden in eine Struktur namens ZLEAD_STRUCT_2 gespeichert, die durch den Tablesparameter IM_LEAD_TAB des Funktionsbausteins repräsentiert wird. Die Daten aus dieser Struktur kommen später auf den Benutzeroberflächen zur Anzeige. Aufgrund dieser Informationen werden folgende Javaklassen und xml Dateien generiert:

Abbildung 26: Generierte xml Dateien für die Datenfelder eines ABAP Funktionsbausteins in einem Web Dynpro Projekt

Die Abbildung listet sechs verschiedene xml Dateitypen auf für die Datenfelder aus den Tabellen CRMD_ORDERADM_H und CRMD_LEAD_H, die in der Struktur zusammengefasst sind: `gsimpletype`, `dtsimpletype`, `gstructure`, `dtstructure`, `xlf` und `dtlogicalddic`. Die `gsimpletype` und `gstructure` Dateien liegen im Verzeichnis **`webAppMainDir\gen_ddic\datatypes\`**`edu\fhf\diplom\spg\model\lead\types`. Die `dtsimpletype`, `dtstructure`, `xlf`, und `dtlogicalddic` Dateien liegen in **`webAppMainDir\src\packages\`**`edu\fhf\diplom\spg\model\lead\types`. Die sonstigen Dateitypen `wdmodel` und `wdmodelclass` befinden sich in in **`webAppMainDir\src\packages\`**`edu\fhf\diplom\spg\model\lead`.

Die Dateitypen `gsimpletype`, `dtsimpletype` und `xlf` machen den Hauptteil der Dateien aus dem Listing aus. Ferner fällt auf, dass die Dateinamen aller Dateien dieses Typs abgesehen von der Extension dreimal vorkommen. Z.B. **`Crmt_Process_Type`**`.gsimpletype`, **`Crmt_Process_Type`**`.dtsimpletype`, und **`Crmt_Process_Type`**`.dtsimpletype.xlf`. Dies kommt daher, dass jedes Dateientrippel ein Tabellenfeld einer Datenstruktur aus dem SAP System

beschreibt. Diese drei Dateien beispielesweise beschreiben das Feld ZLEAD_STRUCT2::PROCESS_ TYPE, die Geschäftsvorgangsart, die ursprünglich aus der Tabelle CRMD_ORDERADM_H stammt.

Die Dateien Verzeichnis `webAppMainDir\gen_ddic\datatypes\edu\fhf\diplom\spg\model\lead\types\Zlead_Struct_2.gstructure`, `webAppMainDir\src\packages\edu\fhf\diplom\spg\model\lead\types\Zlead_Struct_2.dtstructure` und `Zlead_Struct_2.dtstructure.xlf` definieren die Struktur `Zlead_Struct_2` als ganzes. D.h. dass dort nochmals alle Felder als xml Elemente aufgezählt werden. Im selben Verzeichnis gibt es noch die Datei `LeadModel.dtlogicalddic`. Diese ist die Verbindung zum Web Dynpro Projekt, da in ihr als xml Elementattribut `name="LeadModel"` der Name der Modelkomponente für den Funktionsbaustein Z_ALL_LEADS_BY_PARTNER_2 und der Name `logicalSystemName="IJA_CRM_META_DEST"` der einen Umgebungsvariable für die Verbindung vom SAP System und dem SAP Web Application Server enthält, hinterlegt sind.

Für eine Modelkomponente eines Web Dynpro Projekts, bzw. einen Funktionsbaustein eines SAP Systems werden neben den xml Dateien auch Javaklassen generiert. Diese befinden sich im Verzeichnis **`webAppMainDir\gen_wdp\packages\`**`edu\fhf\diplom\spg\model\lead`. Für das Lead Model und den Funktionsbaustein sind dies die Z_ALL_LEADS_BY_PARTNER_2 folgenden:

- `LeadModel.java`
- `Z_All_Leads_By_Partner_2_Input.java`
- `Z_All_Leads_By_Partner_2_Output.java`
- `Zlead_Struct_2.java`

Die Namen dieser Javaklassen entsprechen den Knotennamen die unter der Modelkomponente LeadModel der Referenzanwendung zu finden sind:

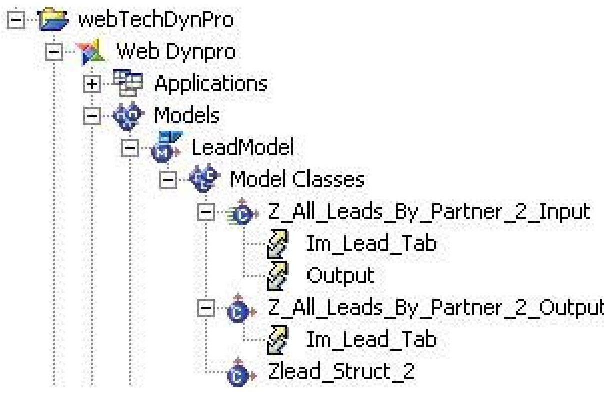


Abbildung 27: Übersicht über eine Modelkomponente im Web Dynpro Explorer

Die Klasse `LeadModel` hat die Funktion, Metadaten über den Funktionsbaustein Z_ALL _LEADS_BY_PARTNER_2 verfügbar zu machen. Dazu gehören:

- Die beiden Umgebungsvariablen für die Verbindung des WAS und des SAP Systems, die der Entwickler bei der Erstellung des Models angegeben hat.
- Die vollklassifizierenden Klassennamen für Klassen der beiden Unterknoten `Z_All_Leads_ By_Partner_2_Input`, `Z_All_Leads_By_Partner_2_Output` und `Zlead_Struct_2`.

Die Klasse `Z_All_Leads_By_Partner_2_Input` dient zur Bereitstellung der Daten aller Import- und Exportparametern sowie aller Tablesparemeter, die der Funktionsbaustein benötigt, um seine Funktionalität ausführen zu können. So kann mittels `Z_All_Leads_By_Partner_2_Input.setIm_Partner(String)` der Wert, den der Benutzer für den Geschäftspartner eingibt, für den die Leadgrunddaten gesucht werden sollen, festgelegt und bei Bedarf mit `Z_All_Leads_By_Partner_2_Input.getIm_Partner()` ausgelesen werden. Ferner können auch mehrere `Zlead_Struct_2` Objekte gesichert werden, die man braucht, wenn z.B. für einen Geschäftspartner mehrere Datensätze gefunden wurden, die jeweils in ein `Zlead_Struct_2` Objekt überführt werden. Zu diesem Zweck gibt es die Methoden:

- `Zlead_Struct_2.getIm_Lead_Tab()`
- `Zlead_Struct_2.addIm_Lead_Tab(Zlead_Struct_2)`
- `Zlead_Struct_2.removeIm_Lead_Tab(Zlead_Struct_2)`
- `Zlead_Struct_2.setIm_Lead_Tab(com.sap.aii.proxy.framework.core.AbstractList)`

`Z_All_Leads_By_Partner_2_Output` bietet keine zusätzliche Funktionalität und ist für die Entwicklung nicht von Bedeutung.

Die letzte generierte Klasse `Zlead_Struct_2` ist eine getter/setter Klasse für die Attribute der Struktur ZLEAD_STRUCT_2 für den Tablesparameter IM_LEAD_TAB des Funktionsbausteins Z_ALL_LEADS_ BY_PARTNER_2.

4.2.4 Der Controller und View Context

Ohne einen Datenspeicher können in einer Web Dynpro Applikation keine Daten vom SAP System zur Benutzeroberfläche und umgekehrt gesendet werden. Diese Aufgabe erfüllt der Context, den jede Controllerkomponente und jede Viewkomponente besitzt. Damit der gesamte Datenaustausch funktionieren kann, müssen die Contexte von Views und Controller miteinander verbunden werden. Erreichbar ist der Context einer View oder eines Controllers im SAP Developer Studio über den Tabreiter Context, nachdem man auf den Component Controller, oder einer Viewkomponente eines Controllers im Web Dynpro Explorer geklickt hat. Der Controller Context der Referenzanwendung hat folgenden Inhalt:

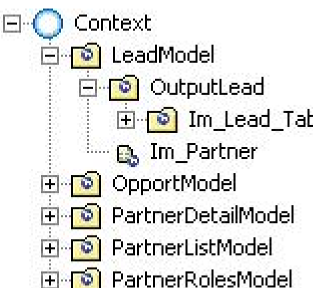


Abbildung 28: Der Controller Kontext einer Controller Komponente eines Web Dynpro Projekts

Es ist zu sehen, dass der Controller Context alle Modelkomponenten beinhaltet. Am Beispiel des `LeadModels` lassen sich die zwei Parameter des Funktionsbausteins Z_ALL_LEADS_ BY_PARTNER_2, der Importparameter Z_ALL_LEADS_ BY_PARTNER_2::IM_PARTNER und der Tablesparameter Z_ALL_LEADS_BY_PARTNER_2::IM_LEAD_TAB erkennen. Beim Aufklappen des Tablesparameters sind alle Datenfelder der Struktur `Zlead_Struct_2` zu finden, die später in einer der Views der Controller Komponente evtl. angezeigt werden sollen.

Angelegt werden die Inhalte eines Contextes eines Controllers oder einer View mit einem Rechtsklick auf den Wurzelknoten oder einen bereits vorhandenen Unterknoten und der Wahl des Kontextmenupunktes `New`. So bekommt man die Auswahl zwischen den Knotentypen `Value Node`, `Value Attribute`, `Model Node`, `Model Attribute`, und `Rekursive Node`. Mit den beiden erst genannten Knotentypen lassen sich beliebige Hierarchien von Knoten und Blättern erstellen, wenn die gespeicherten Daten nicht im Zusammenhang mit Funktionsbaustein – Modelkomponenten stehen.

Für Modelkomponenten wie das `LeadModel` sind der `Model Node` und der `Model Attribute` Knotentyp vorgesehen. Für das Anlegen des `LeadModels` als Beispiel für alle Funktionsbausteinmodels geht man folgendermaßen vor:

1. Anlegen einer `Model Node` und Eingabe dessen Namens
2. Rechtsklick auf die neu erstellte `Model Node` und Auswahl des Menupunktes `Edit Model Binding`
3. Klick auf den langen `LeadModel` Knopf falls die `LeadModel` Inhalte noch nicht ausgewählt sind.
4. Auswahl des **ersten** Unterknotens von `LeadModel` und Klick auf `Next`.

5. Checken des `Im_Partner` Knotens und von `Im_Lead_Tab` unter **`Output`**, nicht unter `Lead Model`. Somit werden alle Unterpunkte von `Im_Lead_Tab` automatisch gecheckt. Es ist nicht notwendig für alle Felder der Struktur `Zlead_Struct_2 Model Attributes` anzulegen.
6. Ergebnis: Die `LeadModel` Modelkomponete wurde dem Controller Context hinzugefügt.

Vorsicht Falle:
Es ist unbedingt empfehlenswert bei Schritt fünf den `Output` Knoten umzubenennen, weil für jede andere Modelkomponente ebenso ein `Output` Unterknoten angelegt wird. Es ist jedoch verboten, dass Unterknoten von verschiedenen Modelkomponenten denselben Namen haben. Für das `LeadModel` eigenet sich beispielsweise der Name `OutputLead`.

4.2.5 Das View Design und das Navigation Handling

Das View Design und das Navigation Handling besteht aus drei Teilen:

1. Erstellen der View Komponenten und deren Navigationsverknüpfungen
2. Das Einrichten des View Contextes nach dem selben Schema wie beim Controller Context
3. Das Designen der Benutzeroberflächen mit UI Elementen und das Verbinden von UI Elementen, z.B. Textfeld, mit dem View Context zur Datendarstellung.

Über den sogenannten `Navigation Modeler` werden View Komponenten der Web Dynpro Anwendung hinzugefügt. Zu öffnen ist dieser über einen `Windows` Knoten der erstellten Controller Komponente über das Kontextmenu und dem Menupunkt `Open Navigation Modeler`. Nachdem alle Views auf dem `Navigation Modeler` platziert wurden, kann mit deren Verknüpfung begonnen werden, die über `Inbound`- und `Outbound Plugs` das Navigationsverhalten zwischen den Views bestimmen. Auf der nachstehenden Abbildung sind alle Views und die blauen `Inbound`- sowie die roten `Outbound Plugs` der Referenzanwendung abgebildet.

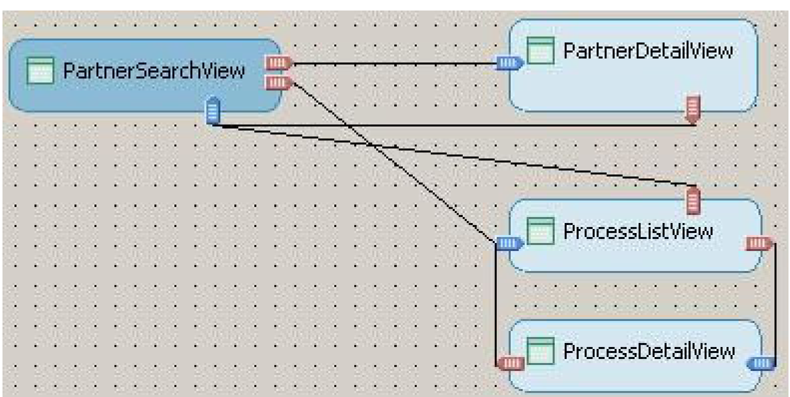


Abbildung 29: Der Navigation Modeler des SAP Developer Studios zur Modellierung des Navigationsverhaltens zwischen Benutzeroberflächen

Um von einer Seite nach einem Klick auf einen Button auf eine andere zu gelangen, ist auf der Ausgangsseite ein `Outbound Plug` und auf der Zielseite ein `Inbound Plug` anzubringen. Nach dem Verbinden des `Outbound Plug` mit dem `Inbound Plug` muss noch eine `Action` beim Ausgangsseiten – View angelegt werden. Dies geschieht über den `Actions` Tab der View. Zu erreichen ist die Viewkonfiguration durch einen Doppleklick auf das richtige Viewelement unter

dem Views Knoten der erstellen Controllerkomponente. Beim Anlegen der Action im Dialogfenster ist der Name der Action anzugeben, und bei der Drop Down Box die Auswahl des Outbound Plugs, das beim Klick auf den Button gefeuert werden soll. Der finale Schritt muss durch die folgenden drei Aufgaben erledigt werden:

1. Klick auf das Layout Tab der Ausgangsseite
2. Auswahl des Button UI Element
3. Auswahl im Properties Fenster rechts unten im Developer Studio von Event/onAction und Auswahl der eben erstellten Action

Sind all diese Schritte erledigt, wird beim Laufen der Anwendung durch den Klick auf den Button auf der Ausgangsseite, auf die Zielseite verzweigt. In der Javaklasse der Viewkomponente wird für die Action eine Methode angelegt, in der der Entwickler Funktionalität implementieren kann.

Um Daten aus dem View-, bzw. Controller Kontext in Textfelder, Tabellen, Trees oder anderen hierfür vorgesehenen UI Elemente auf einer Viewkomponente anzeigen zu können, ist lediglich ein einfacher Schritt notwendig. Auf den Viewseiten der Referenzanwendung ist bei einem Klick auf ein Textfeld im Properties Fenster das Attribut value und aus dem View Controller Kontext das gewünschte Element auszuwählen. Dies führt dazu, dass wenn im Controller Kontext für dieses Attribut ein Wert gespeichert wurde, dieser dann auch im Textfeld dargestellt wird. Bei einer Tabelle läuft dies analog über ein Rechtsklick auf das Tabellen UI Element und der Auswahl des Menupunktes Create Binding.

Vorsicht Falle:

Beim Erstellen von Inbound- und Outbound Plugs ist auf das Wählen sinnvoller Namen zu achten. Bei der Referenzanwendung gibt es folgende Namenskonvention:

Inbound Plug: From<StartView>**I**

Outbound Plug: From<StartView>**O**

Beispiel: Von der Verzweigung von PartnerSearchView nach PartnerDetailView:

Inbound Plug: FromPartnerSearchI

Outbound Plug: FromPartnerSearchO

4.2.6 Grundlegende Codeimplementierung

Um nach dem Erstellen der Controller- und View Kontexte auch das Füllen mit Daten aus den angelegten Modelkomponenten, sprich Funktionsbausteinen zu gewährleisten, sind in den Javaklassen der Controller- und View Komponenten noch ein paar Zeilen Code zu implementieren. Für den Controller muss für das LeadModel folgendes notiert werden:

```
public class ControllerComponent
{
  //...

  public ControllerComponent(IPrivateControllerComponent wdThis)
  {
    this.wdThis = wdThis;
    this.wdContext = wdThis.wdGetContext();
    //...
  }

  public void wdDoInit()
  {
    Z_All_Leads_By_Partner_2_Input leads = new Z_All_Leads_By_Partner_2_Input();
    wdContext.nodeLeadModel().bind(leads);
    //...
  }
```

```
  public void executeProcessListModel()
  {
    try
    {
      String imPartner = wdContext.currentPartnerListModelElement().getIm_Bu_Partner();
      wdContext.currentLeadModelElement().setIm_Partner(imPartner);
      wdContext.currentLeadModelElement().modelObject().execute();
      //...
    }
  }

  //...
}
```

Über das `wdContext` Objekt kann auf die Inhalte des Controller Kontext zugegriffen werden. In der `ControllerComponent.wdDoInit()` Methode wird ein `Z_All_Leads_By_Partner_2_ Input` angelegt, das genau der generierten Javaklasse `webAppMainDir\gen_wdp\packages\ edu\fhf\diplom\spg\model\lead\Z_All_Leads_By_Partner_2_Input.java` entspricht. Dieses Objekt muss dem `LeadModel` Knoten des Kontext bekannt gemacht werden, damit die Daten, die später aus dem Funktionsbaustein Z_ALL_LEADS_ BY_PARTNER_2 ausgelesen werden unter die Strukturen des `LeadModel` Knoten kopiert werden können.

Dazu wird über das `wdContext` Objekt der `LeadModel` Knoten ausgewählt: `wdContext.node LeadModel()` und über `ILeadModelNode.bind(Z_All_Leads_By_Partner_2_Input)` der `LeadModel` Knoten an das `Z_All_Leads_By_Partner_2_Input` Objekt gebunden.

Damit sind die Vorraussetzungen geschaffen, um in der Methode `ControllerComponent. executeProcessListModel()` das Ausführen des Funktionsbausteins Z_ALL_LEADS_BY_ PARTNER_2 mit `wdContext.currentLeadModelElement().modelObject().execute()` anzustoßen, sodass die Resultate, sprich die Leaddaten, die dieser Funktionsbaustein selektiert in den Controller Kontext gespeichert werden. Da dieser Baustein einen Importparameter benötigt, wird dieser über das `wdContext` Objekt gesetzt, der dann ebenfalls bis zum SAP System und zum Funktionsbaustein weitergereicht wird. Die Methode `ControllerComponent.execute ProcessListModel()` wird aufgerufen, wenn der Benutzer auf einen Button klickt, dessen `Action` Callbackmethode als Funktionalität den Aufruf dieser Methode beinhaltet, siehe im folgenden Listing `PartnerSearchView.onActionProcessList(IWDCustomEvent)`.

Auf der Viewausgangskomponente die auf die Leaddaten Darstellungskomponente verzweigt ist dieser Code zu schreiben, damit das Tabellen UI Element der Zielseite die Leaddaten anzeigt:

```
public class PartnerSearchView
{
  //...

  public PartnerSearchView(IPrivatePartnerSearchView wdThis)
  {
    this.wdThis = wdThis;
    this.wdContext = wdThis.wdGetContext();
  }

  //...

  public void onActionProcessList(com.sap.tc.webdynpro.progmodel.api.IWDCustomEvent wdEvent)
  {
    wdThis.wdGetControllerComponentController().executeProcessListModel();
    wdThis.wdFirePlugPartnerSearchProcessListO();

  }
  //...

}
```

4.3 Der Aufbau des Frameworks

Oberflächlich betrachtet ist der SAP Web Dynpro Webentwicklungsframework eine große Anzahl von Java Archiv `.jar` Dateien. Der Aufbau dieses Frameworks soll dahingehend erfolgen, dass untersucht wird, aus welchen Bausteinen und Komponenten er besteht und wie diese zu welchem Zweck zusammenspielen. Beim Aufbau der einzelnen Komponenten und bei deren Zusammenwirken ist eine feiner granulierte Betrachtung notwendig. Dabei muss auf Klassenebene heruntergebrochen werden, um die interne Funktionsweise darzustellen. Zum Kern des Web Dynpro Frameworks gehören:

_webdynpro_clientserver.jar	_webdynpro_admin.jar
_webdynpro_model_dynamicrfc.jar	_webdynpro_basesrvc.jar
_webdynpro_modelimpl.jar	_webdynpro_container.jar
_webdynpro_progmodel.jar	_webdynpro_csf.jar
_webdynpro_services.jar	_webdynpro_model_dynamicrfc.jar
_webdynpro_spi.jar	_webdynpro_monitor.jar
_webdynpro_serverimpl.jar	_webdynpro_pdfobject.jar
_webdynpro_clientimpl.jar	_webdynpro_pluginimpl.jar
_webdynpro_runtime_repository.jar	aii_proxy_rt.jar
_webdynpro_runtime_repository_pmr.jar	

Tabelle 22: Die Java Archive aus denen Web Dynpro besteht

Der Fokus der Betrachtung des Web Dynpro Webframeworks liegt auf der architekturtechnischen Basis der Frameworkklassen- und Interfaces für die Model-, View- und Controllerkomponente, auf denen die generierten Klassen aufbauen, mit denen der Programmierer bei der Applikationsentwicklung arbeitet.

4.4 Die Modelkomponente

Die Modelseite des Web Dynpro Frameworks besteht aus zwei Teilen, genauer zwei Klassenhierarchien. Die erste Hierarchie endet mit der generierten Klasse `Z_All_Leads_By_Partner_2_Input` der Web Dynpro Referenzanwendung. Die zweite schließt mit `LeadModel`. Natürlich sind die Namen der Klassen, die das Ende der jeweiligen Hierarchie bilden, immer anders je nachdem wie bei Hierarchie eins der Funktionsbaustein heißt und wie in Fall zwei der Wurzelknoten für den dem `Z_ALL_LEADS_BY_PARTNER_2` Funktionsbaustein ensprechenden Controller und View Kontextes der Web Dynpro Anwendung benannt wurde:

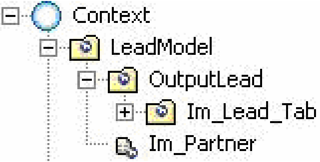


Abbildung 30: Detailliertere Ansicht eines Modelobjekts im Kontext

Basis der beiden Hierarchien sind die Interfaces des Pakets `com.sap.tc.cmi.model`. Die Schnittstellen `ICMIModelClass`, `ICMIModelClassExecutable`, `ICMIQuery` und `ICMIGenericModelClass` schreiben die Basisfunktionalitäten der Klassenhierarchie vor, deren Ende die Klasse `Z_All_Leads_By_Partner_2_Input` bildet. Für die Klasse `LeadModel` gibt es nur ein Wurzelinterface `ICMIModel`. Auffällig ist bei all den Interfaces das Prefix `ICMI`. Das erste `I` kenn-

zeichnet jedes Interfaces als solches. Die Abkürzung CMI steht für Common Model Interface. Klassen die selbst oder über ihre Basisklasse und deren Hierarchie letztendlich diese Schnittstellen implementieren, sind Bestandteile des CMI.

→ Siehe Abbildung 31 in Anhang 5 D

Über das ICMIModelClass Interface sind für eine Modelkomponente wichtige Informationen verfügbar. Mit ICMIModelClass.associatedModel() ist direkter Zugriff auf das Model ICMIModel selbst möglich und mit ICMIModelClass.associatedModelClassInfo() stehen Metadaten als ICMIModelClassInfo Objekt zur Verfügung. Dieses gehört zum Paket com.sap.tc.cmi.metadata und wird später untersucht. ICMIModelClassExecutable besitzt in seiner wesentlichen Aufgabe als signing Interface nur die Methode ICMIModelClassExecutable.execute(). Es dient zur Kennzeichnung von Modelklassen, die ausführbar sind, d.h. dass in ihrer execute Methode mit einem SAP System oder einem Web Service kommunizieren, um von dort persistente Daten zu holen, oder zu modifizieren. Die Methoden des ICMIQuery dienen einer implementierenden Modelklasse zum Ausführen einer Suchabfrage und zum Erhalt der Ergebnisse, sowie ebenfalls Metadaten: ICMIQuery.execute(), ICMIQuery.getResult(), und ICMIQuery.associatedResultInfo(). Ein das ICMIGenericModelClass implementierendes Modelobjekt ist eine Variante eines ICMIModelClass Modelklasse. Um auf Daten und Attribute eines Models zuzugreifen, wird der Java Reflection Mechanismus verwendet. Durch die Implementierung von ICMIGenericModelClass benutzt das Web Dynpro Framework die Methoden ICMIGenericModelClass.getAttributeValue(String) und ICMIGenericModelClass.setAttributeValue(String) durch die Controller und Context Komponenten, die in Kapitel 4.6 behandelt werden.

Auf Seiten der Hierarchie vom ICMIModel bis zum LeadModel stellt dieses Interface zwei Methoden bereit: ICMIModel.associatedModelInfo(String) und ICMIModel.createModelObject(String), wobei letztere noch überschrieben mit einem Class Parameter vorhanden ist. Der Methodenname createModelObject lässt darauf schließen, dass Instanzen, die diese Schnittstelle implementieren, Fabrikklassenfunktionalität haben, die als einzige ICMIModelClass Objekte erzeugen können.

Die beiden Hierarchien besitzen darüber hinaus eine ganze Reihe von Helferklassen und Klassen für zusätzliche Aufgaben, die im folgenden vorgestellt werden.

4.4.1 Die Hierarchie bis Z_All_Leads_By_Partner_2_Input

Der in dieser Hierarchie oberste Baustein ist die Klasse AbstractType aus dem Paket com.sap.aii.framework.core.

→ Siehe Abbildung 33 in Anhang 5 D

Seine Hauptfunktionen sind durch die Implementierung der Schnittstelle com.sap.aii.framework.core.BaseType und dessen wichtigste Methodentypen BaseType.toXml(...) und BaseType.fromXml(...) bestimmt, die jeweils mehrfach überschreiben mit verschiedenen Parametervarianten vorkommen. Mit den BaseType.toXml(...) Methoden und deren privaten Hilfsmethoden AbstractType.renderElement(...), AbstractType.renderElements(...) und AbstractType.renderSimpleType(String, String, String, String, Hashtable) erfolgt die Generierung der xml Dateien unter Zuhilfenahme des com.sap.aii.framework.core.BaseTypeDescriptor für und anhand des Funktionsbausteins Z_ALL_LEADS_BY_PARTNER_2. Siehe dazu Kapitel 4.2.3.

Für den umgekehrten Weg benutzt AbstractType die beiden BaseType.fromXml(...) Methoden. Sprich von den generierten xml Dateien zur Javaklasse Z_All_Leads_By_Partner_2_Input. Die Klasse GenerationInfo ist eine einfache getter Klasse für das Generierungsdatum und die Version der zu generierenden Modelklasse.

→ Siehe Abbildung 34 in Anhang 5 D

Die nächste Hierarchiestufe, die von AbstractType erbt, ist die Klasse AiiModelClass, die keine für das Verständnis des Aufbaus der Modelklassenhierarchie wichtige zusätzliche Funktionalität enthält. Anders ist dies bei der nächsten Ableitung DynamicRFCModelClass, die AiiModelClass erweitert. DynamicRFCModelClass implementiert die eingangs erwähnten Schnittstellen ICMIGenericModelClass, ICMIModelClass, und ICMIModificationCount. Aus dem Paket kommen weitere Interfaces hinzu, die hier zur Konkretisierung kommen. Es sind IWDModelClassRegistrable und IWDModelClassChangeTracking aus dem Paket com.sap.tc.webdynpro.progmodel.model.api, sowie com.sap.tc.webdynpro.modelimpl.dynamicrfc.IWDDynamicRFCModelClass.

Die von ICMIModificationCount zu realisierende Methode ICMIModificationCount.modCount() dient zur Verwaltung von Änderungen an von DynamicRFCModelClass abgeleiteten Modelobjekten. Die Realisierung des IWDModelClassRegistrable Interfaces befähigt eine DynamicRFCModelClass Objekt, sich mit seinem korrespondierenden ICMIModel mittels eines Schlüssel Objects zu verbinden, bzw. zu registrieren und zu deregistrieren. Dazu implementiert DynamicRFCModelClass die Methoden IWDModelClassRegistrable.register(Object) und IWDModelClassRegistrable.unregister(Object).

Die Methoden der Schnittstelle IWDModelClassChangeTracking protokollieren im Fall vn Änderungen an einem DynamicRFCModelClass Objekt welche Art von Änderung stattgefunden hat. Es können sich entweder Attribute eines Models geändert haben, oder die Rollenzugehörigkeit eines Benutzers der auf einer Benutzeroberfläche Daten eines Modelobjektes abfragt. Je nach Rollen des Benutzers werden ihm bestimmte Informationsbestandteile des Models zur Ansicht gewährt ober aber vorenthalten. Z.B. erhält der Vertriebsleiter mit der Rolle Sales Manager im Gegensatz zum Vertriebsmitarbeiter mit der Rolle Sales Agent Daten die gesamte Filiale oder den Bereich betreffend, die der Vertriebsmitarbeiter bei Klick auf eine entsprechende View Komponente nicht zu sehen bekäme. Für Modeländerungen von Attributen gibt es die drei Methoden IWDModelClassChangeTracking.markAttributeChanged(String), IWDModelClassChangeTracking.markAttributeUnChanged(String), und IWDModelClassChangeTracking.isAttributeChanged(String).

Analog zu den Attributen lauten die Methoden für Rollenänderungen: ModelClassChangeTracking.markRoleChanged(String), ModelClassChangeTracking.markRoleUnChanged(String), und ModelClassChangeTracking.isRoleChanged(String). Der String Parameter legt das jeweilige geänderte Attribut, bzw. die Rolle fest.

Das sechste und letzte Interface, das DynamicRFCModelClass implementiert heißt IWDDynamicRFCModelClass. IWDDynamicRFCModelClass.modelInstanceId() liefert jedem DynamicRFCModelClass Derivat eine eineindeutige Identifikationsnummer, IWDDynamicRFCModelClass.scope() bringt den Gültigkeitsbereich des Modelklassenobjekts. Es gibt die folgende Scopes, bzw. Gültigkeitsbereiche in der Klasse WDModelScopeType: COMPONENT_SCOPE, APPLICATION_SCOPE, TASK_SCOPE, NO_SCOPE. Unter TASK_SCOPE versteht man einen einzelnen Request/Response Zyklus. Diese Gültigskeitsbereiche sind mit denen des Servlet APIs.

Bis jetzt gibt für ein ModelClass Objekt noch keine Möglichkeit Kontakt zu einer externen Datenbasis, wie z.B. ein SAP System aufzunehmen und von dort Daten zu beziehen. Hierfür existiert eine weitere Spezialisierung der DynamicRFCModelClass Klasse, die DynamicRFCModel ClassExecutable.

→ **Siehe Abbildung 35 in Anhang 5 D**

Diese Möglichkeit der Verbindungsaufnahme zu einem SAP System wird durch die Interfaces ICMIModelClassExecutable und ICMIQuery vorgegeben. Besonderes Interesse gilt der Methode DynamicRFCModelClassExecutable.execute(). Diese ist die Schnittstelle zur generierten Modelklasse, die direkt von DynamicRFCModelClassExecutable erbt, in diesem Fall Z_All_Leads_By_Partner_2_Input. Diese ruft die abstrakte Methode DynamicRFCModel ClassExecutable.doExecute() auf, die Z_All_Leads_By_Partner_2_Input implementieren muss.

4.4.2 Die Hierarchie bis `LeadModel`

Die Spitze der Klassenhierarchie bis zur generierten LeadModel Klasse bildet Abstract Proxy. Diese ist mit nur einer wichtigen Delegationsmethode AbstractProxy.send(BaseType requestType, String namespace, String interfaceName, String methodName, BaseType responseType) sehr schlank. Jedoch besitzt diese Klasse eine ganze Reihe funktional wichtiger Attribute aus seinem Paket com.sap.aii.proxy.framework.core: JcoRuntimeMetaDataImpl, MessageSpecifierImpl, BaseProxyDescriptor, Descriptor, BaseProxyDescriptor und GenerationInfo.

→ **Siehe Abbildung 38 in Anhang 5 D**

Die Delegationsmethode AbstractProxy.send(BaseType requestType, String name space, String interfaceName, String methodName, BaseType responseType) ist die zentrale Schnittstelle für die Kommunikations mit einem SAP System zum Aufruf von Funktionsbausteinen über das JCo API. Die einzelnen Stationen, die dieser Aufruf nimmt werden kurz dargestellt:

1. AbstractProxy besitzt eine private und statische inner class JcoProxyHelper mit der Methode JcoProxyHelper.send(AbstractProxy proxy, BaseType requestType, String namespace, String interfaceName, String methodName, BaseType responseType).
2. Diese leitet einfach an die gleiche Methode der Klasse JcoProxy weiter.
3. Über den BaseProxyDescriptor des jeweiligen Derivats von AbstractProxy wird der Name des auszuführenden ABAP Funktionsbausteins ermittelt: JcoBaseProxy Descriptor.getAbapFunctionName(String).
4. Als nächstes, damit über das JCo die Kommunikation zum SAP System funktionieren kann, sind ein JCO.Request und ein JCO.Response Objekt zu erzeugen. Der JCO.Request kommt von JcoMarshaler.marshalRequest(String functionName, BaseType request, BaseType response, BaseTypeDescriptor faultMessageDesc, int encoding).
5. Die JCO.Response kommt durch den Aufruf von JCO.Client.execute(JCO.Request). Das JCO.Client Objekt kommt vom MessageSpecifierImpl des AbstractProxys und dessen Methode MessageSpecifierImpl.getJcoClient(). Durch den Aufruf der JCO.

Client.execute(JCO.Request) Methode sind dann die Resultate des ausgeführten ABAP Funktionsbausteins verfügbar.
6. Die JCO.Response muss noch geunmarshalled werden: BaseType = JcoMarshaler. unmarshalRecord(BaseType, JCO.Response).
7. Das erhaltene BaseType Objekt ist dann dem AbstractProxy entsprechende Abstract Type Objekt, das finale Derivat ICMIModel und ICMIModelClass, also LeadModel und Z_All_Leads_By_Partner_2_Input.

Das Attribut BaseProxyDescriptor von AbstractProxy, existiert in zwei Varianten: JcoBaseProxyDescriptor und XmlBaseProxyDescriptor, da diese beiden Klassen das erst genannte Interface implementieren. JcoBaseProxyDescriptor hat die Funktion einer Wrapperklasse um Metainformationen und Ausführungsergebnisse eines ABAP Funktionsbausteins bereit zu stellen. XmlBaseProxyDescriptor wird als Wrapper für eine SOAP Webservice Response verwendet.

MessageSpecifier, bzw. MessageSpecifierImpl ist ebenfalls ein Wrapper. Das gewrappte Objekt ist JCO.Client. MessageSpecifierImpl stellt neben einer get und set Methode noch die Möglichkeit bereit, ein JCO.Client Objekt aus einem java.io. ObjectInputStream zu erzeugen, und ein bestehendes JCO.Client Objekt in ein Object InputStream zu überführen, mit dem dann z.B. eine Serialisierung möglich ist. JcoRuntime MetaDataImpl implementiert JcoRuntimeMetaData., besitzt aber für im Rahmen dieser Betrachtungen keine wichtige Funktionalität.

AiiModel bildet die zweite Hierarchiestufe von AbstractProxy zu LeadModel. Es implementiert ein einziges aber sehr wichtiges Interface IWDDynamicRFCModel.

→ **Siehe Abbildung 38 in Anhang 5 D**

IWDDynamicRFCModel befindet sich im selben Paket wie AbstractProxy. Es steht über ICMIModel in direkter Verbindung zu CMI:

→ **Siehe Abbildung 36 in Anhang 5 D**

Durch IWDDynamicRFCModel und seiner Verbindung zu CMI ist die Funktionalität von AiiModel bestimmt. Die Methoden die von IWDDynamicRFCModel oder seinen Superinterfaces IWDModel und IWDModelRegistry stammen, sind in AiiModel alle als abstract deklariert. Die Hauptaufgaben dieser Klasse sind:

- die Verwaltung eines JCO.Client Objekts, also Connection Management zu einem SAP System. Dazu gehört wiederum: das Beenden von Verbindungen und das Löschen von damit benötigten Ressourcen mit AiiModel.cleanupResources(), das Überprüfen von Synchronität der Verbindung zwischen Client und Server, sprich zwischen Browser und SAP WAS mit AiiModel.verifyClientIsConsistentWithMetadataConnection(JCO. Client) und AiiModel.getCurrentlyUsedJcoClient().
- In der Funktion des Connection Management stellt AiiModel eine Implementierung der Schnittstelle IWDJCOClientConnection bereit, die eine JCo Verbindung repräsentiert, die von Web Dynpro Applikationen verwendet werden. In dieser Eigenschaft arbeitet die IWDJCOClientConnection Implementierung eng mit dem JCO.Client Objekt zusammen und stellt Methoden bereit, die sämtliche Informationen zum jeweiligen SAP System liefern.
- die Verwaltung eines RFCMetadataRepository Objekts. RFCMetadataRepository ist eine Schnittstellenklasse für Metadaten von Objekten des JCo APIs, wie JCO.Repository, JCO.ParameterList, JCO.Function und JCO.Table. Ferner stellt RFCMetadata Repository auch Datentypumwandlungen zur Verfügung von JCo Integerflags zu Class Objektdatentypen wie String.class, Date.class, u.a.. Von JCo Integerflags zu JCo

Datentypsbezeichnern wie z.B. "com.sap.mw.jco.JCO.TYPE_STRING" für String oder "com.sap.mw.jco.JCO.TYPE_BYTE" für ein Byte. Von JCo Integerflags zu XML Schema Datentypen, die für Webservices wichtig sind. Z.B. "http://www.w3.org/2001/XMLSchema:string" oder "http://www.w3.org/2001/XMLSchema:base64Binary".

Die von AiiModel abgeleitete nächste Konkretisierungsstufe ist DynamicRFCModel.

→ Siehe Abbildung 39 in Anhang 5 D

Als erstes fällt auf, dass diese Klasse alle Methoden implementiert, die vom Interface IWDDynamicRFCModel und dessen Basisinterfaces IWDModel und IWDModelRegistry kommen, die in AiiModel realisiert werden. Die zusätzlichen Schnittstellen die DynamicRFCModel implementiert, IWDModel und IMaintainableModel, bringen die neuen Methoden IWDModel.getModelScopeType() und IMaintainableModel.modelMaintainer(). Die erste Methode gibt ein WDModelScopeType Objekt zurück, die zweite eine AbstractModel Maintainer Konkretisierung. Realisiert wird AbstractModelMaintainer durch eine inner class DynamicRFCModelMaintainer innerhalb von DynamicRFCModel. Sie dient zur Verwaltung des Scopes, d.h. eines WDModelScopeType Objekts, also des Gültigkeitsbereichs eines Modelobjekts.

In der Konkretisierungsstufe DynamicRFCModel kommen die Metadaten für Modelobjekte zur Geltung. Das repräsentierende Objekt ist com.sap.tc.webdynpro.modelimpl.dynamicrfc.metadata.DRFCModelInfo. Die strukturelle Basis dieses Objekts liegt im Paket com.sap.tc.cmi.metadata.

→ Siehe Abbildung 32 in Anhang 5 D

Die wichtigesten Bausteine in Verbindung mit DRFCModelInfo sind com.sap.tc.cmi.metadata.ICMIModelInfo, com.sap.tc.cmi.metadata.ICMIAbstractInfo und die abstrakte Klasse com.sap.tc.webdynpro.modelimpl.dynamicrfc.metadata.AbstractInfo. ICMIModelInfo definiert drei Arten des Objekts CMIModelType: RFCAdapter, Reverse Engineered, d.h. ein aus einem XMI File importiertes Model, und GenericModel. Modelobjekte, die einen ABAP Funktionsbaustein abbilden, sind vom CMIModelType RFCAdapter. DRFCModelInfo, die ICMIModelInfo realisiert, definiert noch einen weiteren Typ DynamicRFCAdapter. Merkwürdig an DRFCModelInfo ist, dass nur drei der Methoden des Interfaces ICMIModelInfo zum ICMIRelationRoleInfo Objekt, bzw. dessen Implemenierung DRFCRelationRoleInfo mit Funktionalität implementiert sind: DRFCModelInfo.getModelClassInfos(), DRFCModelInfo.iterateModelClassInfos() und DRFCModelInfo.getModelClassInfo(String name). Die anderen Methoden, die eine java.io.Collection oder einen java.io.Iterator zurückliefern, geben nur null zurück, haben also keine Implementierung.

In ihrer Funktion also zentrale Metadatenklasse hat DRFCModelInfo auch eine Verbindung zu den Metadatenbausteinen des JCo. Die besagten JCo Metadatenobjekte sind: IStructure, IDataType, IField und IMetaData aus dem Paket com.sap.mw.jco. Um sich diese Objekte zu besorgen nutzt DRFCModelInfo das Interface IBroker, das durch com.sap.tc.webdynpro.services.datatypes.core.DataTypeBroker konkretisiert wird.

Die ICMIModelInfo Schnittstelle ist die Wurzel aller Laufzeitmetadaten eines CMI Modelobjekts. Es liefert für ein Derivat von DynamicRFCModel zugeordnete ICMIModelClassInfo Objekte, überprüft ob ein Modelobjekt eine Rollenbeziehung zu einem anderen Modelobjekt besitzen kann. Implementierungstechnisch folgt für diese Fähigkeit, dass solch einem Modelobjekt ICMIRelationRoleInfos zugeordnet sind. Eine ICMIRelationRoleInfo enthält eine ICMIModelClassInfo Repräsentation, eines diesem Model zugeorneten Modelobjekts und die

eigentliche Beziehung ICMIRelationRole. Diese verweist mit ICMIRelationRole.getModel Info() auf das ICMIModelInfo Derivat, also dem Äquivalent zu ICMIModelClass.

Die Realisierung von ICMIRelationRoleInfo DRFCRelationRoleInfo besitzt ein CMI Cardinality Objekt, das die Art der Beziehung zweier verknüpfter Modelobjekte widergibt. Die Beziehungsarten sind 0..1, 0..*, 1..1, 1..*.

4.5 Die Controllerkomponente

Die Controllerkomponente ist das Bindeglied zwischen der Modelkomponente und den Views, die die rohen Daten des Models in einer übersichtlichen Form darstellen sollen. Der Controller steuert dabei, welche Daten eines Models dargestellt werden dürfen, und ist für die Kommunikation von Datenänderungen, sprich Modeländerungen an Viewkomponenten zuständig. Auch bei Benachrichtigung von Änderungen an die Modelkomponenten durch Benutzerinteraktion auf einer View kommt der Controller zum Einsatz. Der Controller ist also dafür verantwortlich, die Modelkompontente(n) immer in einem konsistenten Zustand zu halten.

In Web Dynpro besteht die Controllerkomponente aus drei Teilen. Erstens dem Context, der als reiner Datenspeicher einen Teil der Modeldaten beinhaltet, die bei einer Benutzerinteraktion auf einer Viewkomponente von Bedeutung sind. Welche Daten dies genau sind, und deren Beschaffung, also das Aufrufen oder Ausführen eines Models, sprich eines ABAP Funktionsbausteins und das anschließende Füllen des Contexts mit den erhaltenen Daten aus dem ausgeführten Model, ist die zentrale Funktion des Controllers.

Der Component Teil der Controllerkomponente in Web Dynpro ist eine dem Controller übergeordnete Instanz, die mehrere Controller und deren Contexte und Viewkomponenten zusammenfasst.

Auf diesen drei Bausteinen baut die Generierung der Klassen auf, die für die Implementierung der Controllerkomponente notwendig sind. Wie diese Klassen mit den drei Basis Teilen Context, Controller und Component verbunden sind und, wie deren Aufgabenverteilung organisiert ist wird in Abschnitt 4.5.3 behandelt.

Um die Basis des architekturtechnischen Aufbaus der Controllerkomponente für die Umsetzung dieser Funktionalitäten darzulegen, werden die Klassen des Web Dynpro Frameworks und deren Zusammenspiel sowie deren Verknüpfungen untersucht. Dies erfolgt in den Abschnitten 4.5.1 und 4.5.2. Mit dieser Basis erfolgt im nächsten Schritt die darüberliegende Ebene der generierten Javaklassen für die Controllerkomponente, die in jedem Web Dynpro Projekt erstellt werden, und mit denen der Entwickler die Funktionalität der Anwendung realisiert. Eine weitere wichtige Funktion ist das Abfangen und Bearbeiten von Benutzerinteraktionen.

4.5.1 Die Basis: Context, Controller und Component

Die Basis dieser drei Teile bilden wie auch bei der Modelkomponente Javainterfaces. Die Interfaces für Context, Controller und Component sind IWDContext, IWDController, und IWDComponent aus dem Paket com.sap.tc.webdynpro.progmodel.api. Diese Interfaces dienen neben der Kennzeichnung des jeweiligen Typs auch der Verknüpfung der drei Controllerbestandteile untereinander. So schreibt die Schnittstelle IWDContext für die Context Implementierungsklasse die Methode IWDContext.getController() vor, mit der der Controller feststeht, der für die Dateninhalte des Context verantwortlich ist. Wichtig sind auch die Methoden IWDContext.getRootNode() und IWDContext.getRootNodeInfo() mit denen das

Wurzelelement der Baumstruktur eindeutig identifiziert ist und von wo aus der gesamte Contextinhalt, also alle weiteren Unterelemente rekursiv erreichbar sind.

Da die `Context` Implementierungsklasse für die Erfüllung ihres Zwecks mehr Funktionalität braucht, gibt es das Interface `IContext`. Dessen Methoden drehen sich um zwei Aspekte: Erstens, der Ereignisbehandlung für das Hinzufügen, das Verändern und Löschen von Knotenelementen und deren Attribute. So kann durch spätere Implementierungsklassen, siehe 4.5.3 auf dynamische Contextmodifizierungen während der Laufzeit der Anwendung reagiert werden. Ermöglicht wird dies durch die `IContext` Methoden, die mit fireXYZ beginnen und mit XYZListener enden. Zweitens die Verbindung zum `Component` Teil der Controllerkomponente mit `IContext.getContextManager()`, die ein `Component` Objekt, das die Schnittstelle `IContextManager` implementiert, zurückliefern muss.

`IWDController` setzt für `Controller` die Methoden `IWDController.getContext()` und `IWDController.getComponent()` voraus, um einen `Controller` mit seinem zu verwaltenden `Context` und seinem übergeordneten `Component` Objekt zu verbinden. Durch die Implementierung von `IController` durch die Klasse `Controller` wird die erste Verbindung zu Kapitel 4.5.3 sichtbar: `IController.ExternalInterface()` und `IController.getPublicInterface()`. Die Schnittstellen `IWDExternalControllerInterface` und `IWDPublic ControllerInterface` sind die Basisinterfaces für die generierten Controllerkomponentenklassen. Für seine anfangs erwähnte Funktion der Benutzerinteraktionsbearbietung stellt `IController` gleich mehrere Methoden zur Verfügung: `IController.getActionInternal()`, `IController.createAction()`, `IController.iterateAction()` und `IController.invokeEventHandler()`. Wichtig für Kapitel 4.5.3 ist die Methode `IController.init()`, in der der Entwickler in einer generierten Controllerklasse Funktionalität für die Verbindung von Modelkomponenten zum Controller implementiert.

Die beiden Klassen `Controller` und `Context` sind reine Implementierungsklassen für ihre Interfaces wie das folgende Schaubild zeigt.

→ **Siehe Abbildung 40 in Anhang 5 D**

Die Klasse `Component` realisiert zwei Schnittstellen, `IContextManager` und `IWDComponent`. Da auch jede Viewkomponente einen Kontext besitzt, der ausgewählte oder auch die gesamte Datenstruktur des Controller Kontextes besitzt, hat `Component` über `IContextManager.iterateViewContexts()` die Kontrolle über die Datenspeicher der Views. Mittels `IContextManager.addChangeListener(IContextChangeListener)` und `IContextManager.removeContextChangeListener(IContextChangeListener)` werden `NodeInfo` Objekte einem Event Notification Mechanismus hinzugefügt, der diese Objekte benachrichtigt, wenn sich Änderungen an Kontextstrukturen ereignen, die aus einer Hierarchie von `Node` Obkjeten bestehen. Dazu mehr in Kapitel 4.5.2.

`IContextManager.getTypeBroker()` sorgt für die Verfügbarkeit eines `DataTypeBroker` Objekts. `IWDComponent` stellt die Verbindung zu anderen wichtigen Objekten sicher. Dazu gehören: `IWDWindowManager`, `IWDComponentInfo` und `IWDMessageManager`.

Zusätzlich zu den Methoden aus den beiden Interfaces besitzt `Component` noch eine ganze Reihe anderer Funktionalitäten:

- Das Speichern und Erzeugen von `Controller` Objekten. Sprich Fabrikfunktionalität für diese Objekte oder Derivaten davon
- Das Speichern von Viewkomponenten und Event Handlern. Das Zuordnen von Event Handlern zu `Controller` Objekten.

- Das Verwalten eines MessageManagers. Dieser ist wichtig für das Anzeigen von Nachrichten für den Benutzer der Viewkomponenten. Er kann auch für Debugzwecke verwendet werden.

→ **Siehe Abbildung 41 in Anhang 5 D**

Die im folgenden Diagramm gezeigten Klassen DelegatingComponent und Delegating CustomController sind konkrete Implementierungsschnittstellen, die die generierten Controllerkomponentenklassen per Delegation verwenden um ihre Funktionalitäten ausführen zu können. Welche dies sind und wie genau diese Beziehungen aussehen wird in Kapitel 4.5.3 aufgezeigt.

→ **Siehe Abbildung 42 in Anhang 5 D**

4.5.2 Der Context im Detail: Knoten und Elemente

Alle Daten, die z.B. von einem Model, einem ABAP Funktionsbaustein kommen, und die in einer Web Dynpro Applikation verwendet werden, müssen von einer Controllerkomponente zwischengespeichert werden, damit sie zur Laufzeit der Anwendung zur Verfügung stehen. Der Ort der Speicherung ist der Context eines Controllers. Die Javaobjekte, die die Daten für den Context repräsentieren sind Node, NodeElement und NodeInfo aus dem Paket com.sap.tc. webdynpro.progmodel.context.

Ein Node Objekt entspricht genau einem Knoten in der hierarchischen Baumstruktur des Kontext. Daraus folgt, das eine Node ein oder mehrere Kind Node Objekte haben kann, oder auch keines, sofern es als Blatt Node Objekt eine tiefste Teilhierarchie des Elementbaums ist. Ferner ist jedem Node Objekt sein Elternelement zugeordnet sofern es nicht das Wurzel Node Objekt ist.

→ **Siehe Abbildung 43 in Anhang 5 D**

Das Klassendiagramm verdeutlicht den Aufbau des Node Objekts. Es implementiert wie die anderen zentralen Objekte wie z.B. wie Context oder Controller ein Basisinterface aus dem Paket com.sap.tc.webdynpro.progmodel.api, nämlich IWDNode. Die Methoden von IWD Node dienen hauptsächlich dem Abfragen von Inhalt und Struktur der Node Hierarchie. So ermöglichen es die Methoden IWDNode.movePrevious(), IWDNode.moveElement() und IWDNode.moveNext() durch die Baumhierarchie zu navigieren. IWDNode.getChildNode(String, int), IWDNode.getElementAt(int), IWDNode.getParentElement(), IWDNode. getCurrentElement(), IWDNode.getLeadSelection() ermöglichen es, mit der gegenwärtigen Node verbundene Node und NodeElement Objekte abzufragen.

Ein Node Objekt ist mit den anderen Kontextobjekten NodeElement und NodeInfo verbunden. Das NodeElement kapselt ein Attribut AttributeInfo von denen ein Node Objekt mehrere besitzen kann. Um solch ein Attribut zu bearbeiten, realisiert NodeElement die Methoden IWD NodeElement.getAttributeValue(String attr) und IWDNodeElement. setAttribute Value(String attr, Object value) aus seinem implementierten Interface IWDNode Element. Gespeichert werden die NodeElement Objekte von Node inner classes ElementList, MappedElementList und ModelElementList. Diese drei Klassen bieten über das Speichern hinaus auch noch die Möglichkeit, die Attributelemente des Node Objekts zu sortieren.

Für Metadaten und das Management von Änderungen einer Node ist das NodeInfo Objekt zuständig. Unter Änderungsmanagement versteht man hier das Event Processing, wenn dem Node Objekt dynamisch ein Attribut, also ein NodeElement oder ein Kind Node Objekt,

hinzugefügt wurde. Diese Änderungen müssen an die betroffene Node kommuniziert werden. Dies wird über die Methoden IContextChangeListener.onNodeAdded(IContext, Node Info), IContextChangeListener.onAttributeAdded(IContext, AttributeInfo), IContextChangeListener.onAttributeDataTypeChanged(IContext, AttributeInfo), und IContextChangeListener.onNodeReset(IContext, NodeInfo) der Schnittstelle IContextChangeListener erreicht. Zu den Metadaten gehören auch die Objekte IStructure, ICMIRelationRoleInfo, IDataType, und ICMIModelClassInfo.

→ Siehe Abbildung 44 in Anhang 5 D

4.5.3 Die Umsetzung: Generierte Klassen mit Controllerfunktionalität

Die generierten Klassen der Controllerkomponente bauen allesamt auf den in den beiden vorherigen Kapiteln vorgestellten Basisklassen auf, sprich durch Vererbung, oder sie benutzen diese als Klassenattribute, also per Delegation. Alle Klassen und Interfaces aus den Web Dynpro Projektjavapaketen edu.fhf.diplom.spg.controller.wdp und edu.fhf.diplom.spg.controller haben einen einen gemeinsamen Teilstring in ihrem Klassnenamen: Controller Component. Dies kommt daher, dass es im Web Dynpro Projekt im SAP Developer Studio eine Controllerkomponente mit genau diesem Namen gibt. Siehe dazu die Abbildung 24. Je nach Bezeichnung einer solchen Controllerkomponente im Web Dynpro Projekt durch den Programmierer heissen die generierten Klassen anders. Alle anderen Namensbestandteile der generierten Klassen sind aber immer dieselben. Z.B. der Name des Interfaces IPublic**ControllerComponent**Interface. Hätte der Applikationsentwickler seine Controllerkomponente im Web Dynpro Projekt XYComponent genannt, wäre der Name des IPublicControllerComponentInterface IPublic**XYComponent**Interface. Analog gilt dies für alle anderen Klassen auch, die als Teil ihres Namens ControllerComponent haben.

→ Siehe Abbildung 45 in Anhang 5 D

Der Aufbau der Controllerkomponente der vom Web Dynpro Codegenerator erstellten Klassen für dieses Web Dynpro Projekt wie auch für jedes beliebige andere Projekt, lässt sich in drei Teile gliedern, wovon jedoch nur dem ersten Teil eine entscheidende Rolle zukommt. Sieht man sich die Teile zweitens und drittens an, stellt man fest, dass es dort außer der Delegation an andere Klassen keine neue Funktionalität im Vergleich zum ersten Teil gibt. Die drei Teile der Controllerkomponente sind:

1. Alle Klassen und Interfaces, die in einer Vererbungs- oder Delegationsbeziehung mit **InternalControllerComponent** stehen. Durch die Implementierung von IPrivateControllerComponent besitzt InternalControllerComponent eine Verbindung zum Controller Component Objekt durch IPublicControllerComponent.wdGetAPI(). Durch das Component Objekt ist auch das Controller und Context Objekt verfügbar. Die Inhalte des Context Objekts sind durch IPrivateControllerComponent.getGetContext() erreichbar, da das zurückgelieferte IContextNode Objekt, als Derivat von com.sap.tc.webdynpro.progmodel.context.Node die Wurzel des Kontext ist.

   Die Verbindung zu den dem Controller zugeordneten Modelobjekten wird über IPublicControllerComponent.createContextElement() und die zahlreichen IPublicControllerComponent.createXYZElement(...) und IPublicControllerComponent.currentXYZElement(...) hergestellt. Jeder Knoten des Controller Kontext ist über eine einzelne Methode für das Web Dynpro Framework und für den Entwickler verfügbar.

Wichtig ist auch die Beziehung zwischen `InternalControllerComponent` und `ControllerComponent`. `InternalControllerComponent` besitzt eine Reihe von Delegationsmethoden. D.h., dass für die Methoden `wdDoInit()`, `wdDoExit()`, `wdDoPost Processing(boolean)`, `wdDoBeforeNavigation(boolean)` und bei spezifischen Methoden des jeweiligen Controllers, die entweder durch Codegenerierung auf Basis von Modelobjektstrukturen, oder durch den Entwickler hinzugefügt wurden, lediglich an die Klasse `ControllerComponent` weitergeleitet werden, wo die eigentliche Funktionalität, die der Entwickler implementiert, ausgeführt wird.

Über `DelegatingComponent`, ein Derivat der Controllerkomponentenbasisklasse `Component` werden die Methoden `InternalControllerComponent.wdCreateAction( WDActionEventHandler, String)` und `InternalControllerComponent.wdCreate NamedAction(WDActionEventHandler, String, String)` von der Schnittstelle `IPrivateControllerComponent` implementiert.

2. Die Klassen und Schnittstellen um **`InternalControllerComponentInterface`**. Sie haben lediglich Delegations- und Schnittstellenfunktion. Delegiert werden Methodenaufrufe an die Klassen `com.sap.tc.webdynpro.progmodel.generation.DelegatingCustom Controller` und an die generierte Klasse `ControllerComponentInterface`, die keine Klasse erweitert und auch kein Interface implementiert. Ihr Name ist je nach Namensgebung Controllerkomponente des Entwicklers im Web Dynpro Projekt unterschiedlich, wie alle anderen generierten Klassen auch die den Teilstring `ControllerComponent` in ihrem Klassennamen haben.
3. Die beiden wichtigsten Dinge rund um die Klasse `InternalControllerComponent InterfaceView` sind die Methoden `IpublicControllerComponentInterfaceView. wdGetAPI()` die ein `IWDViewController` Objekt zurückliefert und `IViewDelegate.wd CreateUITree()`. von der ein `IWDViewElement` Objekt erzeugt wird, das die Wurzel der GUI Elementhierarchie einer Benutzeroberfläche abbildet und somit durch Rekursion den gesamten Inhalt einer Viewseite repräsentiert. Die Objekte `IWDViewController` und `IWDViewElement` sind Bestandteile der Viewkomponente aus dem folgenden Kapitel 4.6

Die Klasse `com.sap.tc.webdynpro.progmodel.context.Context` ist der zentrale Bestandteil der Controllerkomponente, die mit einer Baumhierarchie von `com.sap.tc. webdynpro.progmodel.context.Node` Objekten den gesamten Inhalt von Modelobjekten speichert, die dem Applikationsanwender dargestellt werden muss. Da sich die Struktur und der Inhalt eines ABAP Models, also die Ergebnisse eines ABAP Funktionsbausteins immer unterscheiden, sieht natürlich der Inhalt des `Context` und damit der Inhalt und Aufbau der `Node` Objekthierarchie immer anders aus. Um diesem Umstand Rechnung zu tragen, muss ebenfalls ein `Node` Derivat generiert werden, das die ABAP Models in ihren individuellen Eigenschaften abbildet.

Die Klasse, die je nach Model immer neu generiert wird, ist eine inner class von `IPublic ControllerComponent` heisst `IContextNode`. Ihr zugeordnet ist wie aus Kapitel 4.5.2 bekannt ein Derivat von `NodeElement`, `IContextElement`.

4.6 Die Viewkomponente

Die Viewkomponente dient zur Darstellung der Daten der Modelkomponente und wird von der Controllerkomponente dahingehend gesteuert, einen Teil oder auch gesamte Inhalte von einzelnen oder verschiedenen Modelobjekten anzuzeigen.

Mittels der Viewkomponente kann der Benutzer ausser der reinen Datenansicht auch Änderungen an Modelobjekten vornehmen. Die Änderungen werden an den Controller gesendet, der die Modelkomponente auf den aktuellen Stand bringt. Ergeben sich Änderungen an Modeldaten aus anderen Gründen, wird die Viewkomponente von der Controllerkomponente darüber in Kenntnis gesetzt. Die Viewkomponente ist damit die Schnittstelle für den Anwender, um Informationen aus den Modeldaten zu erhalten, die für seine Zwecke von Bedeutung sind und seinerseits selbst Daten zu ändern, zu erweitern oder zu löschen.

Um den Aufbau der Viewkomponente offenzulegen, werden zunächst wie bei der Model- und Controllerkomponente auch, die Basisinterfaces betrachtet, die die wesentliche Funktionalität der implementierenden Klassen festlegen. Darauf folgt eine Analyse dieser Klassen und eine Übersicht über die Web Dynpro GUI Bibliothek. Der Schluss dieses Kapitels widmet sich der für das individuelle Web Dynpro Projekt im SAP Developer Studio generierten Klassen der Viewkomponente, die dem Design der Benutzeroberflächen Rechnung tragen und die Verbindung zur Controllerkomponente herstellen.

4.6.1 Die Basisschnittstellen der Viewkomponente

Das Basisinterface aller Viewkomponentenklassen ist das Interace `com.sap.tc.webdynpro.progmodel.api.IWDView`.

→ Siehe Abbildung 48 in Anhang 5 D

Es erweitert `IWDViewController` und dieses wiederum `IWDController`. Das heisst also, dass eine Viewklasse zumindest Controller*funktionalität* besitzt. Die Methode `IWDViewController.firePlug(IWDOutboundPlugInfo, Map parameters)` ermöglicht es, einer Viewkomponente auf eine andere View zu navigieren, wie es durch das vom Entwickler festgelegten Navigationsverhalten gewollt ist, siehe Kapitel 4.2.5. Die Methoden der Schnittstelle `IWDView` befassen sich mit dem `Context` der Viewkomponentenklasse. Durch `IWDView.getRootElement()` und `IWDView.getElement()` lässt auf diesen `Context` zugreifen. Welche Daten aus Modelkomponenten in dem `Context` einer Viewkomponente verfügbar sind, ist Aufgabe des Entwicklers beim Design der Benutzeroberflächen.

4.6.2 Derivate der View Klasse

Die abstrakte Klasse `com.sap.tc.webdynpro.progmodel.view.View` implementiert die grundlegenden Funktionalitäten einer jeden Viewkomponente, sprich die Methoden aus den Schnittstellen, die sie implementiert. Bei Betrachtung dieser Methoden fällt auf, dass der eigentliche Kern der Implementierung bei anderen Klassen liegt. Man kann auch hier sagen, dass `View` in diesen Fällen an ihr wichtigstes Attribut, `ViewManager` weiterleitet, bzw. delegiert. Z.B. in den Methoden `View.createElement(Class, String id)`, `View.firePlug(IWDOutboundPlugInfo, Map parameters)`, `View.navigate(IWDOutboundPlugInfo, Map parameters)`, und `View.requestFocus(IWDAction)` und noch bei einigen anderen mehr.

→ Siehe Abbildung 49 in Anhang 5 D

Besondere Bedeutung bezüglich des Kontextmanagements kommt der View Helferklasse ContextBindingManager zu. Unter Kontextmanagement versteht man alles, was durch die Methoden ausgeführt wird, dessen Namen mit Changed enden. Die meisten dieser Methoden sind aus Platzgründen nicht im vorherigen Klassendiagramm aufgeführt:

- View.editablePropertyBindingChanged(ViewElement, String[], String[])
- View.leadSelectionBindingChanged(String[], String[])
- View.multipleNodeAttributeBindingChanged(String[], String[])
- View.multipleNodeBindingChanged(String[], String[])
- View.multipleNodeRangeBindingChanged(MultipleNodeRangeBinding, String[], String[])
- View.ordinaryAttributeBindingChanged(String[], String[])
- View.primaryEventBindingChanged(ViewElement, IWDAction, IWDAction)

Auffallend ist, dass alle Methodennamen das Wort Binding enthalten. Unter diesem Binding versteht man hier das Verbinden von Kontextinhalten mit einem GUI Element. Z.B. lässt sich ein Kontextknoten, der den Namen eines Artikels beinhaltet, mit einem Textfeld auf der Benutzeroberfläche verbinden, damit er diesen anzeigt sobald der Benutzer diese Seite besucht. Die hier aufgeführten Methoden dienen zur Reaktion auf Ereignisse, die sich auf dynamische Änderungen zur Laufzeit der Anwendung beziehen. Die Methode View.primaryEventBinding Changed(ViewElement, IWDAction, IWDAction) beispielsweise behandelt Ereignisse, wenn ein anderes Kontextelement mit einem GUI Element verbunden wird. Da diese Funktionalitäten zu den grundlegenden Anforderungen an jede Viewkomponente gehören, muss die abstrakte Basisklasse View diese bereitstellen. Da aber es aber aus Gründen des guten Softwaredesigns unangebracht wäre, dies alles in View zu implementieren, sind diese Funktionalitäten in die Klasse ContextBindingManager ausgelagert.

Der ContextBindingManager reagiert aber nicht nur auf Binding Änderungen zwischen Kontextknoten und GUI Elementen, sondern auch auf Änderungen bei Kontextinhalten der Viewkomponente. Die Methoden, die der ContextBindingManager dazu implementiert, kommen von dem Interface IContextChangeListener. Es ist das selbe Interface, das auch von ContextInfo ralisiert wird, eine Superklasse von NodeInfo, siehe Kapitel 4.5.2.

Zur Implementierung der Methoden aus den Basisinterfaces von View besitzt View die Attribute IWDViewElement und IWDViewContainer. Das IWDViewElement Klassenattribut ist das Wurzelelement aller GUI Elemente, die sich auf einer Viewkomponente befinden. IWDView Container kennzeichnet eine besondere Eigenschaft eines GUI Elements. Eine GUI Komponente, die diese Schnittstelle implementiert, kann andere GUI Komponenten aufnehmen, ähnlich einem JPanel, oder einem JScrollContainer aus der Java Swing GUI Bibliothek. Mittels solcher GUI Elemente lässt sich eine Benutzeroberfläche in Gebiete aufteilen, wie es z.B. bei Webseiten bekannt ist, mit einem Navigationsmenu links oder oben auf der Seite und mit einem Hauptteil, der die eigtenliche Information beinhaltet.

Wie bei der Controllerkomponente auch gibt es bei der Viewkomponente Derivate von View, die als Schnittstellenklassen von den generierten Javaklassen, die im folgenden Kapitel näher untersucht werden, genutzt werden. Diese Klassen sind DelegatingView und Delegating InterfaceView.

Ein Blick auf diese Klassen zeigt, dass sie sich in ihrer Funktionalität kaum unterscheiden. Beide delegieren Methodenaufrufe an ein Derivat von IViewDelegate. Das Web Dynpro Framework nutzt diese beiden Klassen um letztlich die generierten Viewkomponentenklassen aus Kapitel 4.6.3 aufzurufen, wodurch die eigentliche programmierte Anwendung des Entwicklers funktioniert. Das folgende Schaubild zeigt den Zusammenhang zwischen IViewDelegate und den generierten Viewkomponentenklassen.

→ Siehe Abbildung 50 in Anhang 5 D

`IViewDelegate` ist von `IControllerDelegate` abgeleitet und dieses wiederum von `IWDPublicControllerInterface`. Diese Interfaces zusammenbetrachtet beinhalten alle Methoden, an die die Klassen `DelegatingView` und `Delegating InterfaceView` delegieren. Implementiert werden diese Methoden, sprich das Interface `IViewDelegate` von `Internal PartnerSearchView`. D.h. `DelegatingView` und `DelegatingInterfaceView` delegieren immer an eine generierte Viewkomponentenklasse.

4.6.3 Die Umsetzung: Generierte Klassen mit Viewfunktionalität

Bisher wurde zwar der Ausdruck Viewkomponente oft benutzt, jedoch noch nicht geklärt, was genau darunter zu verstehen ist. Eine Viewkomponente besteht neben den Basisview-klassen– und Interfaces, die in den beiden vorherigen Kapiteln behandelt wurden aus den generierten Javaklassen und Schnittstellen, die für eine Benutzeroberfläche mit samt ihren Inhalten und ihrem Design generiert wurden. Es gibt immer ein Interface und zwei Klassen mit folgenden Namen: die Namen der beiden Klasse ist `InternalXYZ` und `XYZ`. Der Name des Interface lautet `IPrivateXYZ`. Der fixe Teil `XYZ` ist die Bezeichnung, die der Entwickler der Viewkomponente des Web Dynpro Projekts im SAP Developer Studio gegeben hat. Heisst z.B. eine Viewkomponente `PartnerSearchView`, lauten die Klassen- und Interfacenamen `Internal PartnerSearchView`, `PartnerSearchView`, und `IPrivatePartnerSearchView` wie auf folgendem Klassendiagramm zu sehen ist.

→ Siehe Abbildung 51 in Abschnitt 6.5

Das Interface `IPrivatePartnerSearchView` bildet mit inner classes Zugriffsklassen für jeden Hauptknoten den gesamten Viewkontext ab. Für jeden Hauptknoten gibt es in der entsprechenden inner class get und set Methoden für den Zugriff und das Ändern eines jeden Unterelementsknoten. Wie der Viewknotext der `PartnerSearchView` Viewkomponente zeigt, gibt für die zugehörige Benutzeroberfläche drei Hauptknoten, die alle Daten enthalten, die auf dieser View dargestellt werden können.

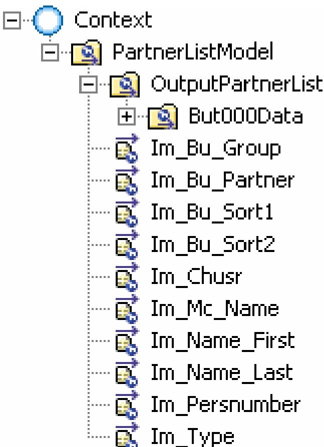


Abbildung 52: Der Kontext der Viewkomponente PartnerSearchView

Einer der Kontexthauptknoten heisst `PartnerListModel`. Im Interface `IPrivatePartner SearchView` wurden hierfür zwei inner classes generiert: `IPrivatePartnerSearchView.`**`IPartnerListModelElement`** und `IPrivatePartnerSearchView.`**`IPartnerListModelNode`**. Die Namen der inner classes für die anderen Kontexthauptknoten bilden sich nach demselben

Schema: IXYZModelElement und IXYZModelNode. XYZ ist hier der Name des Kontexthauptknotens. Die Funktion von IPartnerListModelNode als Ableitung von com.sap.tc.webdynpro.progmodel.context.Node ist zum einen Zugriff auf das IPartnerListModelElement Objekt zu ermöglichen: IPrivatePartnerSearchView.IPartnerListModelNode.currentPartnerListModelElement() und IPrivatePartnerSearchView.IPartnerListModelNode.getPartnerListModelElementAt(int). Zum anderen die Erzeugung eines Node Repräsentationsobjekts eines Kindelements für ein gegebenes Elternelement durch IPartnerListModelNode.createChildNode(NodeInfo childInfo, NodeElement parentElement).

IPartnerListModelElement stellt als Derivat von com.sap.tc.webdynpro.progmodel.context.MappedNodeElement und letztlich von com.sap.tc.webdynpro.progmodel.context.NodeElement die get und set Methoden für die Unterelementknoten des Kontexthauptknotens PartnerListModel bereit. Zwei Beispiele:

Knoten	Methoden
PartnerListModel.Im_Type	String IPartnerListModelNode.**getIm_Type**() IPartnerListModelNode.**setIm_Type**(String)
PartnerListModel.Im_Bu_Partner	String IPartnerListModelNode.**getIm_Bu_Partner**() IPartnerListModelNode.**setIm_Bu_Partner**(String)

Je nach Struktur und Anzahl der Subknoten kann diese inner class sehr groß sein.

Die Klasse die dem Entwickler für seine spezifischen Implentierungen zur Verfügung steht ist PartnerSearchView. Hier legt er durch eigene Methoden fest, auf welche Viewkomponente(n) durch Aktionen verzweigt wird, wie z.B. durch das Klicken auf eine Schaltfläche. Auch das Anstoßen der Ausführung von ABAP Funktionsbausteinen, bzw. Modelkomponenten durch Controller Komponenten, die in das Web Dynpro Projekt importiert wurden, wird hier vorgenommen. Um Zugriff auf den Viewkontext und seine get und set Methoden zu erhalten, besitzt PartnerSearchView als eine Referenz auf IPrivatePartnerSearchView.

Die generierte Klasse InternalPartnerSearchView erweitert IPrivatePartnerSearchView und das für die Verbindung von Basisframeworksklassen wie DelegatingView und DelegatingInterfaceView auf der einen und den generiertes Klassen der Viewkomponente wichtige Interface IViewDelegate auf der anderen Seite. Damit ist diese Klasse der Kommunikationspartner zwischen Framework und den darauf aufsetzenden generierten Klassen für den Entwickler. Hier wird auch die für den Entckler entscheidende Methode IViewDelegate.wdDoInit() durch Delegation an das Klassenattribut PartnerSearchView realisert. So erreicht das Web Dynpro Framework schließlich das Aufrufen der vom Entwickler geleisteten Applikationslogik.

Eine weitere entscheidende Funktion ist die Realisierung der Methode InternalPartnerSearchView.createUITree() aus dem Interface IViewDelegate. Diese liefert ein IWDViewElement zurück, das die Wurzel der UI Elemente ist, die der Entwickler auf der Benutzeroberfläche platziert hat.

4.7 Die Web Dynpro UI Komponentenbibliothek

Wie in Kaiptel 4.6.2 schon angedeutet ist das Interface com.sap.tc.webdynpro.progmodel.api.IWDViewElement die Basis der Web Dynpro UI Komponentenbibliothek. Davon direkt abgeleitet ist com.sap.tc.webdynpro.progmodel.api.IWDUIElement. Siehe dazu auch Abbildung 47 in Abschnitt 4.5. Im Paket com.sap.tc.webdynpro.clientserver.uielib.standard.api befinden sich die Basisinterfaces für jedes einzelne Element der Web

Dynpro UI Bibliothek. Z.B. für eine Schaltfläche IWDButton, für eine Checkbox IWDCheckBox, für einen Radiobutton IWDRadioButton, usw. Die Implementierungsklassen dieser Schnittstellen liegen in com.sap.tc.webdynpro.clientserver.uielib.standard.impl.

Die meisten UI Elementklassen erweitern die abstrakte Klasse UIElement, die das Interface IWDUIElement realisiert. So auch die Klasse ViewContainerUIElement, die IWDView ContainerIElement implementiert. IWDViewContainerUIElement selbst erweitert IWDView Container. Daraus folgt, dass das Attribut IWDViewContainer der View Basisklasse von ViewContainerUIElement initialisiert wird.

4.8 Zusammenfassung

Das Webentwicklungsframework SAP Web Dynpro setzt wie die Open Source Vertreter Struts und JSF auf das Model View Controller Muster. Der Web Dynpro Codegenerator erstellt je nach Aufbau und Struktur von ABAP Funktionsbausteinen Javaklassen, die diesen repräsentieren. Genutzt werden diese Artefakte erstens im SAP Developer Studio als Basis für den Context von Controllerkomponenten und zweitens zum Ausführen von Funktionsbausteinen seitens des Frameworks für die Anwendung.

Die vom Codegenerator erzeugten Klassen für Viewkomponenten basieren auf denen vom Entwickler designten Benutzoberflächen. Über einen Viewkontext, der mit dem Controller Kontext verbunden ist, wird erstens ein Bezug zum Model tier hergestellt, mit dem dann zweitens die Modelinhalte über den Controller für die Viewkomponente darstellbar sind. Für jeden Modelknoten, sprich Datenattribut, gibt es über ein generiertes View- und Controllerartefakt eine Zugriffsmethode, die genau einem Attribut eines ABAP Funktionsbausteins entspricht.

Die Benutzeroberflächenvorlagen im SAP Developer Studio sind reine GUI Element Container. Es befindet sich dort keinerlei Referenzen auf Model- oder Controllerfunktionalität, geschweige denn konkrete Implementierungen. Das reine Aussehen der Viewoberflächen ist in xml Dateien hinterlegt, die genau die vom Viewdesigner erstellten GUI Elementhierarchie der Oberfläche beinhaltet.

Die Verbindung von abstrakten Basisklassen des Web Dynpro Frameworks mit den generierten Klassen von induviduellen Controller- und Viewkomponenten wird über Delegation hergestellt. D.h., dass Basisframeworkklassen Attribute besitzen, die als Schnittstellen von den generierten View- und Controllerklassen implementiert werden, wodurch das Framework in der Lage ist, die konkreten Methoden der jeweiligen generierten Klasse aufruft.

5 Anhang

A Das Struts Framework in Kurzform

Anhang A zeigt den Ablauf der Initialisierungsphase einer Struts Webapplikation.

```
ActionServlet.init()
  ActionServlet.initInternal()
    MessageResources.getMessageResources()
      MessageResourcesFactory.createFactory()
        RequestUtils.applicationClass()
        Class.newInstance()
      MessageResourcesFactory.createResources()

  ActionServlet.initOther()
    ServletConfig.getInitParameter("config")
    ServletConfig.getInitParameter("debug")
    ServletConfig.getInitParameter("convertNull")
    ConvertUtils.deregister()
    ConvertUtils.register(BigDecimalConverter, BigDecimal)
    ...
    ConvertUtils.register(ShortConverter, Short)

  ActionServlet.initServlet()
    new Digester()
    ServletContext.getResourceAsStream("web.xml")
    Digester.parse(web.xml)

  ActionServlet.initModuleConfig(nullPrefix, "/WEB-INF/struts-config.xml")
    ModuleConfigFactory.createFactory()
    ModuleConfigFactory.createModuleConfig()
    ServletConfig.getInitParameter("mapping")
    ModuleConfig.setActionMappingClass()
    ActionServlet.initConfigDigester()
      new Digester()
      Digester.setNamespaceAware()
      Digester.setValidating()
      Digester.setUseContextClassLoader()
      Digester.addRuleSet(new ConfigRuleSet())
      Digester.register(docType, dtd_url)
      ServletConfig.getInitParameter("rulesets")
      RequestUtils.applicationInstance(ruleset)
      Digester.addRuleSet(RuleSet)
    Digester.parseModuleConfigFile()
      ServletContext.getResource(path)
      new InputSource(url)
      ServletContext.getResourceAsStream(path)
      InputStream.setByteStream(InputSource)
      Digester.parse(InputStream)
      ServletContext.setAttribute(Globals.MODULE_KEY + prefix, ModuleConfig)
    ModuleConfig.findFormBeanConfigs()
      FormBeanConfig.getDynamic()
      DynaActionFormClass.createDynaActionFormClass(FormBeanConfig)
```

```
ActionServlet.initModuleMessageResources(ModuleConfig)
  ModuleConfig.findMessageResourcesConfigs()
  MessageResourcesConfig.getFactory()
  MessageResourcesFactory.setFactoryClass(factory)
  MessageResourcesFactory.createFactory()
  MessageResourcesFactory.createResources()
    new PropertyMessageResources()
  MessageResources.setReturnNull(MessageResourcesConfig.getNull())
  ServletContext().setAttribute(MessageResourcesConfig.getKey() + ModuleConfig.
                              getPrefix(), MessageResources)

ActionServlet.initModuleDataSources(ModuleConfig)
  new ServletContextWriter(ServletContext)
  ModuleConfig.findDataSourceConfigs()
  RequestUtils.applicationInstance(DataSourceConfig.getType())
  BeanUtils.populate(DataSource, DataSourceConfig.getProperties())
  DataSource.open()
  DataSource.setLogWriter()
  ServletContext().setAttribute(DataSourceConfig.getKey(),
                              ModuleConfig.getPrefix(), DataSource)
  FastHashMap.put(DataSourceConfig.getKey(), DataSource)

ActionServlet.initModulePlugIns(ModuleConfig)
  ModuleConfig.findPlugInConfigs()
  ServletContext().setAttribute(Globals.PLUG_INS_KEY + ModuleConfig.getPrefix(),
                              PlugIn[])
  PlugIn[i] = RequestUtils.applicationInstance(PlugInConfig.getClassName())
    ...
  BeanUtils.populate(PlugIn[i], PlugInConfig[i].getProperties())
    ...
  PropertyUtils.setProperty(PlugIn[i], "...", PlugInConfig[i])
    ...
  PlugIn[i].init(ActionServlet.this, ModuleConfig)
     ... [evtl. Bsp. ValidatorPlugIn + Subframework erklären]

ModuleConfig.freeze()
Enumeration = ServletConfig().getInitParameterNames()

// jetzt für jede Subapplikation dasselbe wie für die Defaultapplikation
while(Enumeration) {
  ActionServlet.initModuleConfig(prefix, ServletConfig().getInitParameter(name)))
  // z.B.: name = "/WEB-INF/zyx/struts-xyz-config.xml

  ActionServlet.initModuleMessageResources(ModuleConfig)
  ActionServlet.initModuleDataSources(ModuleConfig)
  ActionServlet.initModuleDataSources(ModuleConfig)
  ActionServlet.initModulePlugIns(ModuleConfig)
}

ActionServlet.destroyConfigDigester()
```

B Das JSF Framework in Kurzform

von JSP mit JSF Tags zum HTML

```
CommandButtonTag.doStartTag()
  UIComponentTag.doStartTag()
    UIComponentTag.setupResponseWriter()
    UIComponentTag.encodeBegin()
      UIComponent.encodeBegin()
        UIComponentBase.encodeBegin()
          UIComponentBase.getRenderer(FacesContext)
          Renderer.encodeBegin(FacesContext, UIComponentBase)
            ButtonRenderer.encodeBegin(FacesContext, UIComponentBase)
```

vom Request vom FacesServlet zur Action processing Methode ActionListener. processAction(ActionEvent)

a) ActionEvent Objekterzeugung und Queueing

```
ApplyRequestValuesPhase.execute(FacesContext)
  UIViewRoot.processDecodes(FacesContext)
    UIComponentBase.processDecodes(FacesContext)
      UIComponentBase.decode(FacesContext)
        UIComponentBase.getRendererType()
        UIComponentBase.getRenderer(context)
          ButtonRenderer.decode(context)
            new ActionEvent(UIComponent)
            UIComponent.queueEvent(ActionEvent)
              HtmlCommandButton.queueEvent(ActionEvent)
              UICommmand.queueEvent(ActionEvent)
                UIComponentBase.queueEvent(ActionEvent)
                  UIViewRoot.queueEvent(ActionEvent)
```

b) Ausführen des Events durch die ActionListener Implementierung

```
InvokeApplicationPhase.execute(FacesContext)
  UIViewRoot.processApplication(FacesContext)
    UIViewRoot.broadcastEvents(FacesContext, PhaseId.INVOKE_APPLICATION)
      FacesEvent = eventList(i)
      UIComponent = FacesEvent.getComponent()
      UIComponent.broadcast(FacesEvent)
        UICommmand.broadcast(FacesEvent)
          ActionListener = ListenerList.next()
            ActionEvent.processListener(ActionListener)
              ActionListener.processAction(ActionEvent)
              UIComponent = ActionEvent.getComponent()
              ActionSource = (ActionSource)UIComponent
              MethodBinding = ActionSource.getAction()
              String outcome = MethodBinding.invoke(FacesContext, null)
              NavigationHandler.handleNavigation(FacesContext, outcome)
              FacesContext.renderResponse()
```

```
ViewTag.doEndTag()
  UIComponentBodyTag.doEndTag() // existiert nicht, daher:
    UIComponentTag.doEndTag()
      UIComponentTag.encodeBegin()
        UIComponent.encodeBegin(FacesContext)
          Renderer.encodeBegin(FacesContext, UIComponent)
      UIComponentTag.encodeChildren()
        UIComponent.encodeChildren(FacesContext)
          Renderer.encodeChildren(FacesContext, UIComponent)
      UIComponentTag.encodeEnd()
        UIComponent.encodeEnd(FacesContext)
          Renderer.encodeEnd(FacesContext, UIComponent)
```

C Konfiguration der Verbindung von SAP System und SAP Web Application Server

SAP System

1) Anlegen einer Login Gruppe: Transaktion SMLG
2) Transaktion SM59
   - TCP/IP Verbindungen
     - SLD_NUC
     - SLD_UC
     → Gateway Host: SAP System Server
       Gatewayservice: saggw<SAP Systemnummer> (z.B. 00)

SAP WAS Konfigrationsoberfläche

- Technical Landscape
  - New Technical System
    - Web AS ABAP
      - Web AS ABAP Name (SID): z.B. EBB
      - Installation Number: Siehe SAP GUI: Menu System→ Status
      - Database Host Name: **Name** des SAP Servers, nicht IP
    - Message Server
      - Host Name: z.B. wfr00656
      - Message Port: 36[00]. [00] ist die jeweilige Instanznummer des SAP Systems
      - Login Gruppe: Siehe 1) unter SAP System
    - Central Applicaton Server
      - Host Name
      - Instance Number
    - Application Server
      - Host Name
      - Client Number: <Mandant>
      - aus Liste auswählen. Z.B. R3 Enterprise 47x110

Dateien editieren

1. `C:/Windows/System32/driver/etc/services`
Einträge hinzufügen:

sapms<SAP Systemname>   3600/tcp

2. `C:/Windows/System32/driver/etc/hosts`
Einträge hinzufügen:

<IP SAP Server>   <SAP Systemname>

Kopieren und importieren von folgenden Dateien

Die beiden zip Dateien im Verzeichnis `/usr/sap/<SYS ID>/sys/global/sld/model` in `SLD/Administration/import` der SAP WAS Konfigrationsoberfläche importieren.

- Object Server definieren und starten (in der SAP WAS Konfigrationsoberfläche unter „Server Settings“)

Visual Administrator

Verzeichnis: `/usr/sap/<SYS ID>/JC00/j2ee/admin/go.bat`

- Unter Cluster/Server0.../Services/SLD Data Supplier rechter GUI Teil:
  - 1. HTTP Settings
    - Host: SAP WAS Serverrechner
    - Port: 50 000
    - User: Adminsitrator
    - Password: ...

  - 2. RFC Settings
    - Alles leer lassen
  - 3.
    - Host, Port, User, Password wie 1.

D Screenshots der Klassendiagramme zum Aufbau des Web Dynpro Frameworks

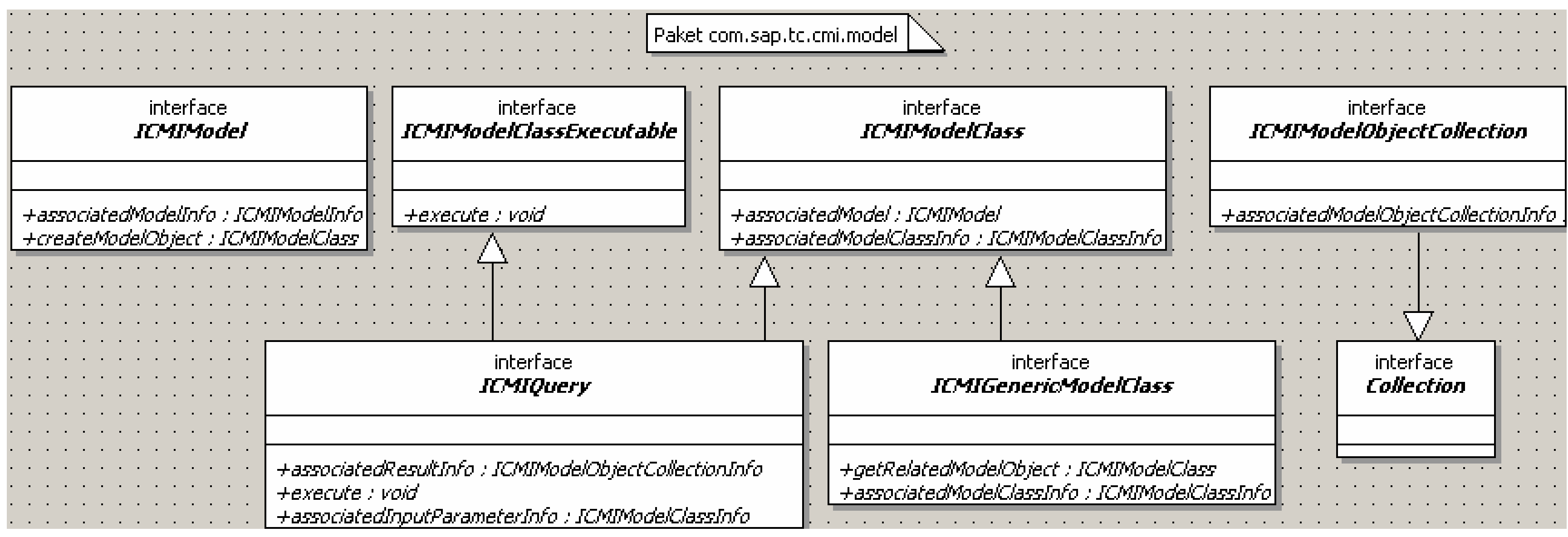


Abbildung 31: Basisinterfaces für CMI Modelklassen aus dem Paket com.sap.tc.cmi.model

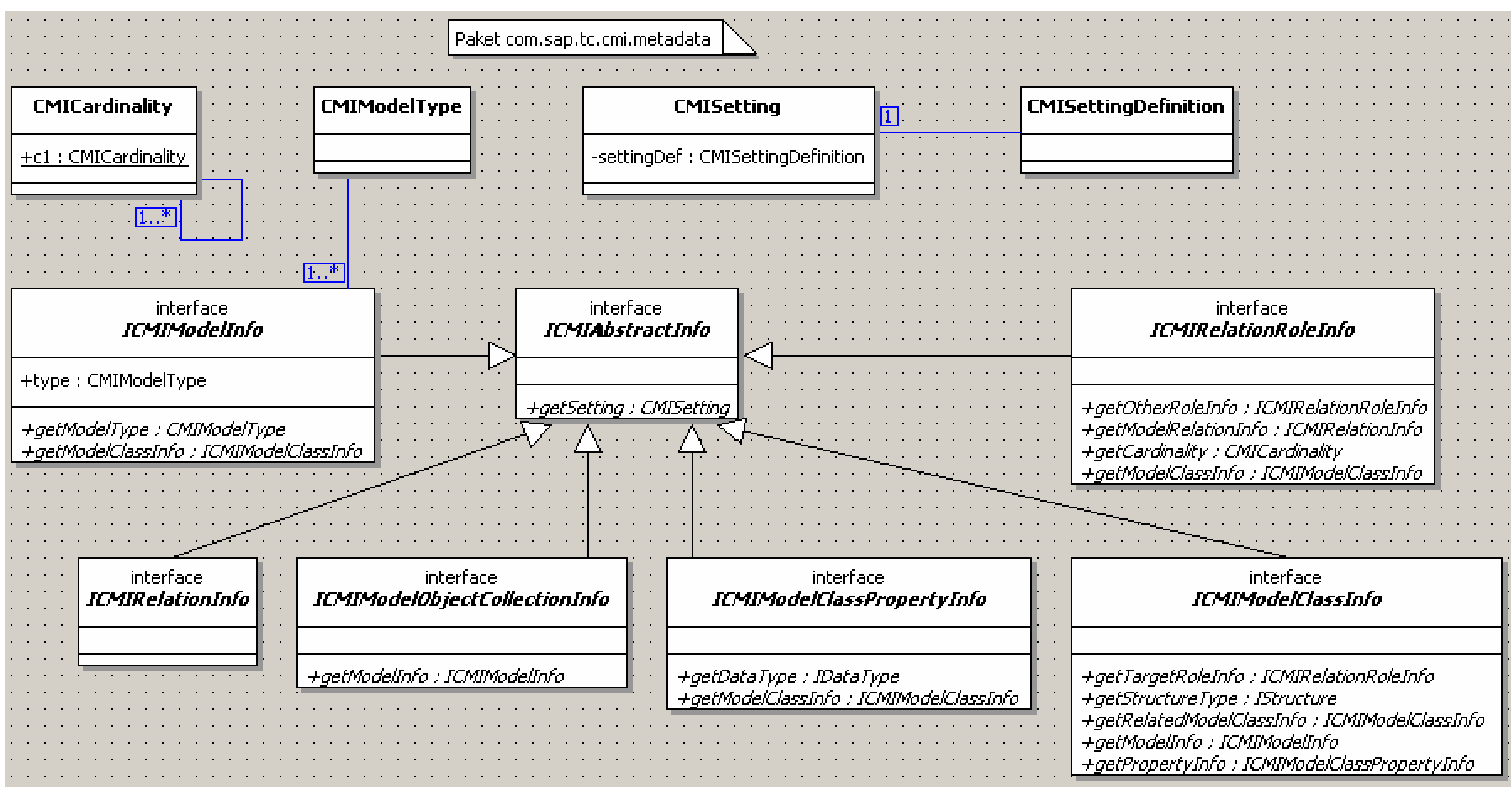


Abbildung 32: Basisinterfaces aus dem Paket com.sap.tc.cmi.metadata für Klassen, die Metadaten über Modelklassen liefern

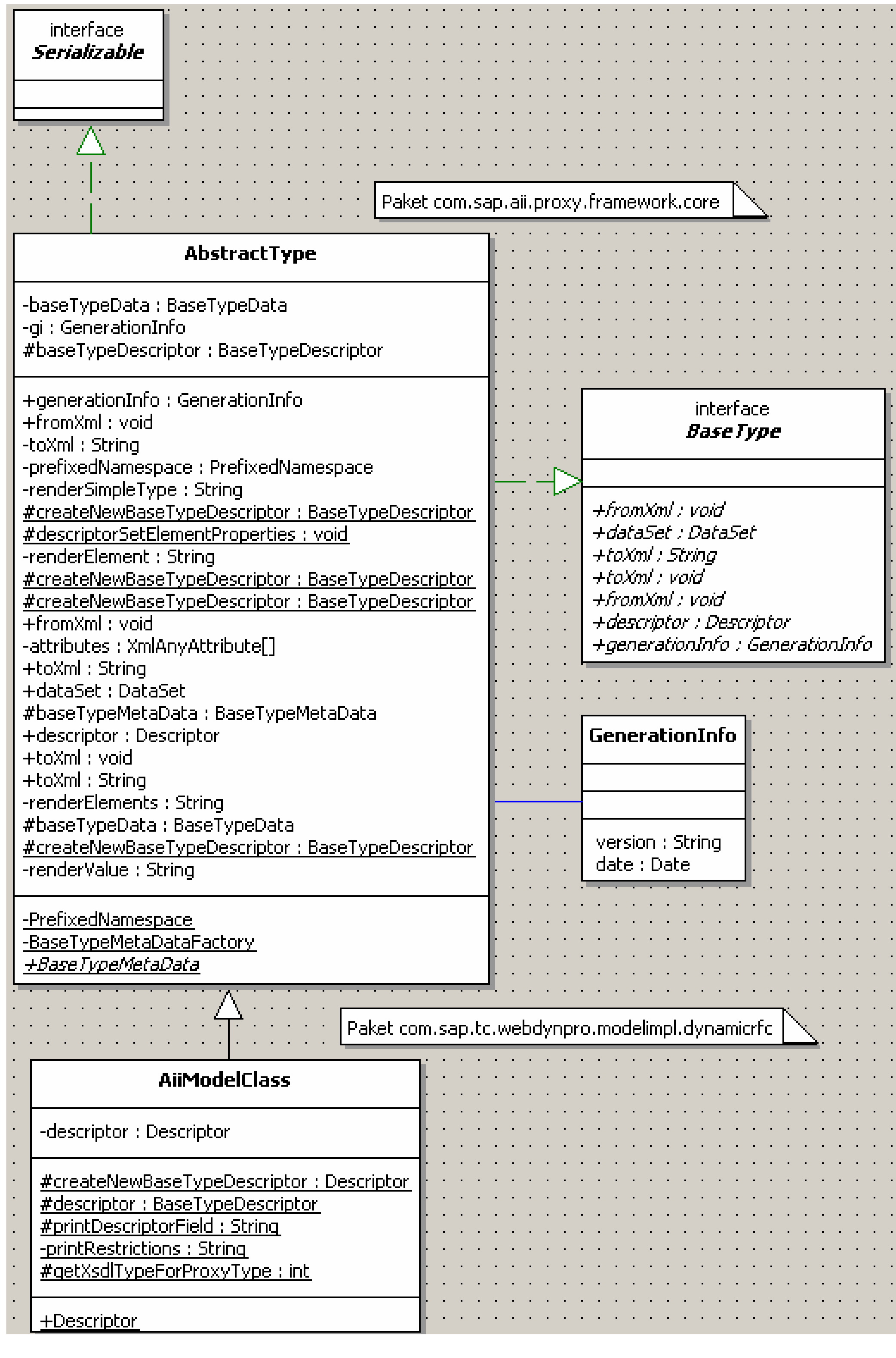


Abbildung 33: Stufe 1 der Modelklassenhierarchie bis Z_All_Leads_By_Partner_2_Input um die Klasse com.sap.aii.proxy.framework.core.AbstractType

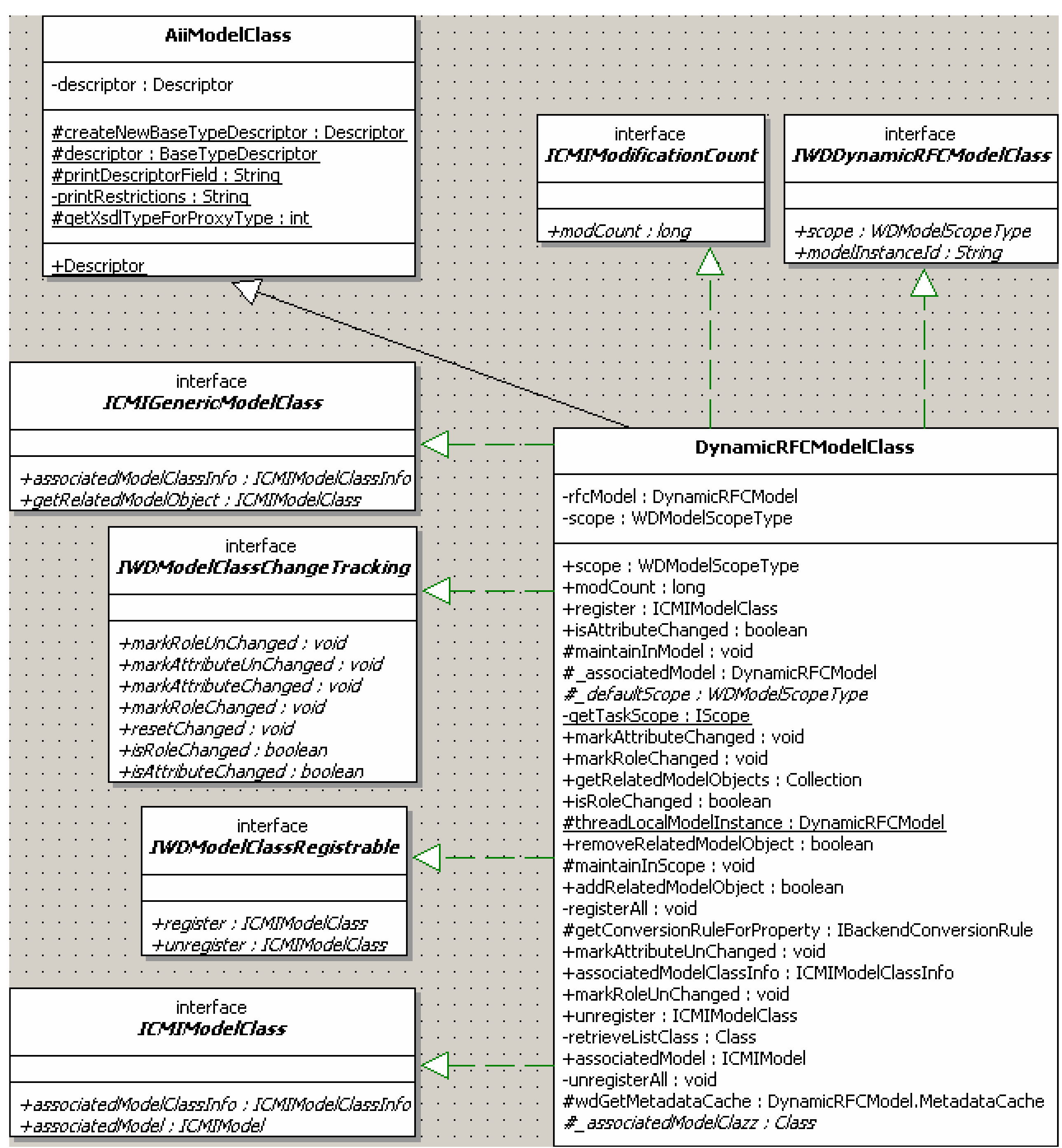


Abbildung 34: Stufe 2 und 3 der Modelklassenhierarchie bis Z_All_Leads_By_Partner_2_Input um die Klassen AiiModelClass und DynamicRFCModelClass

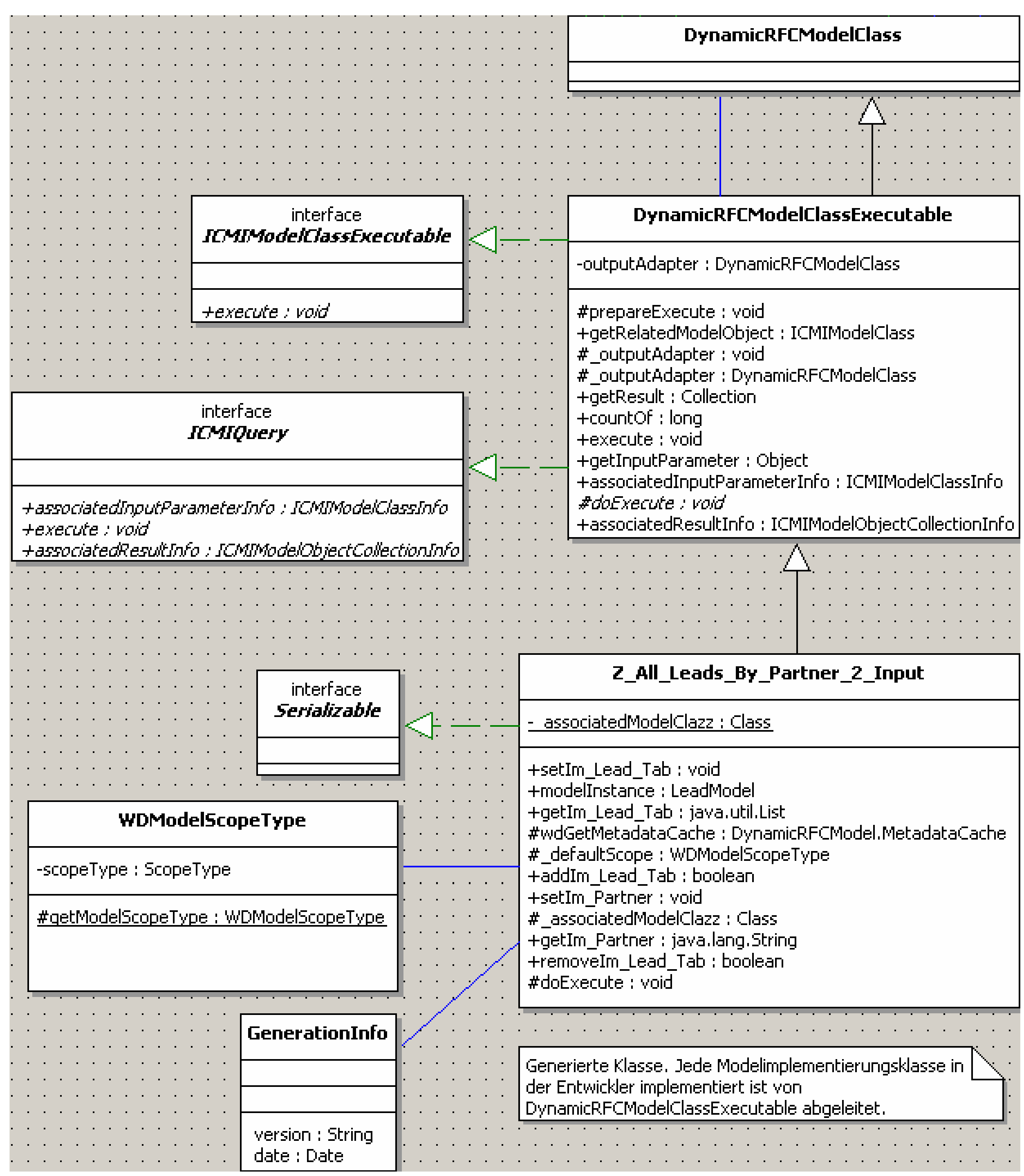


Abbildung 35: Stufe 4 der Modelklassenhierarchie bis Z_All_Leads_By_Partner_2_Input um die Klasse DynamicRFCModelClassExecutable

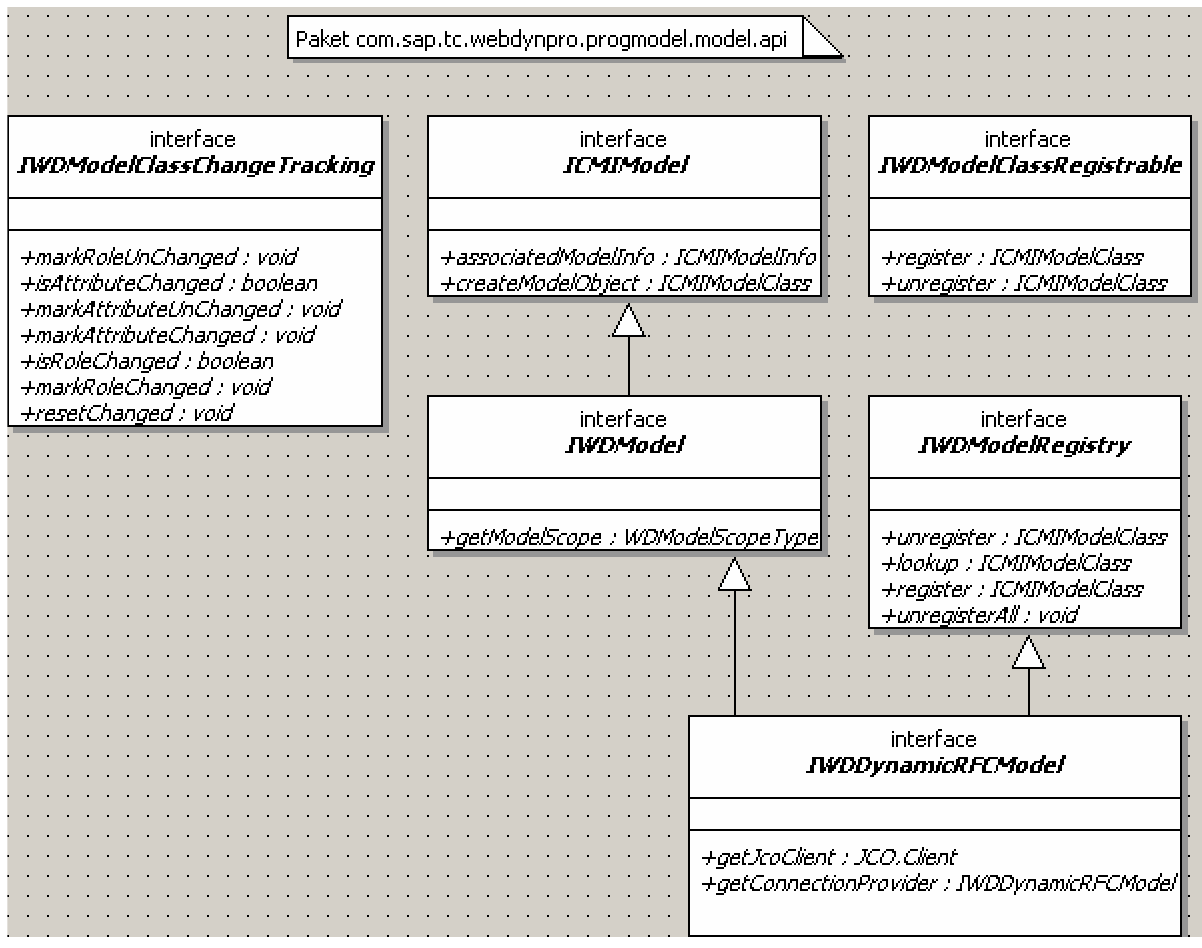


Abbildung 36: Basisinterfaces für ICMIModel Derivate aus dem Paket com.sap.tc.webdynpro.progmodel.model.api

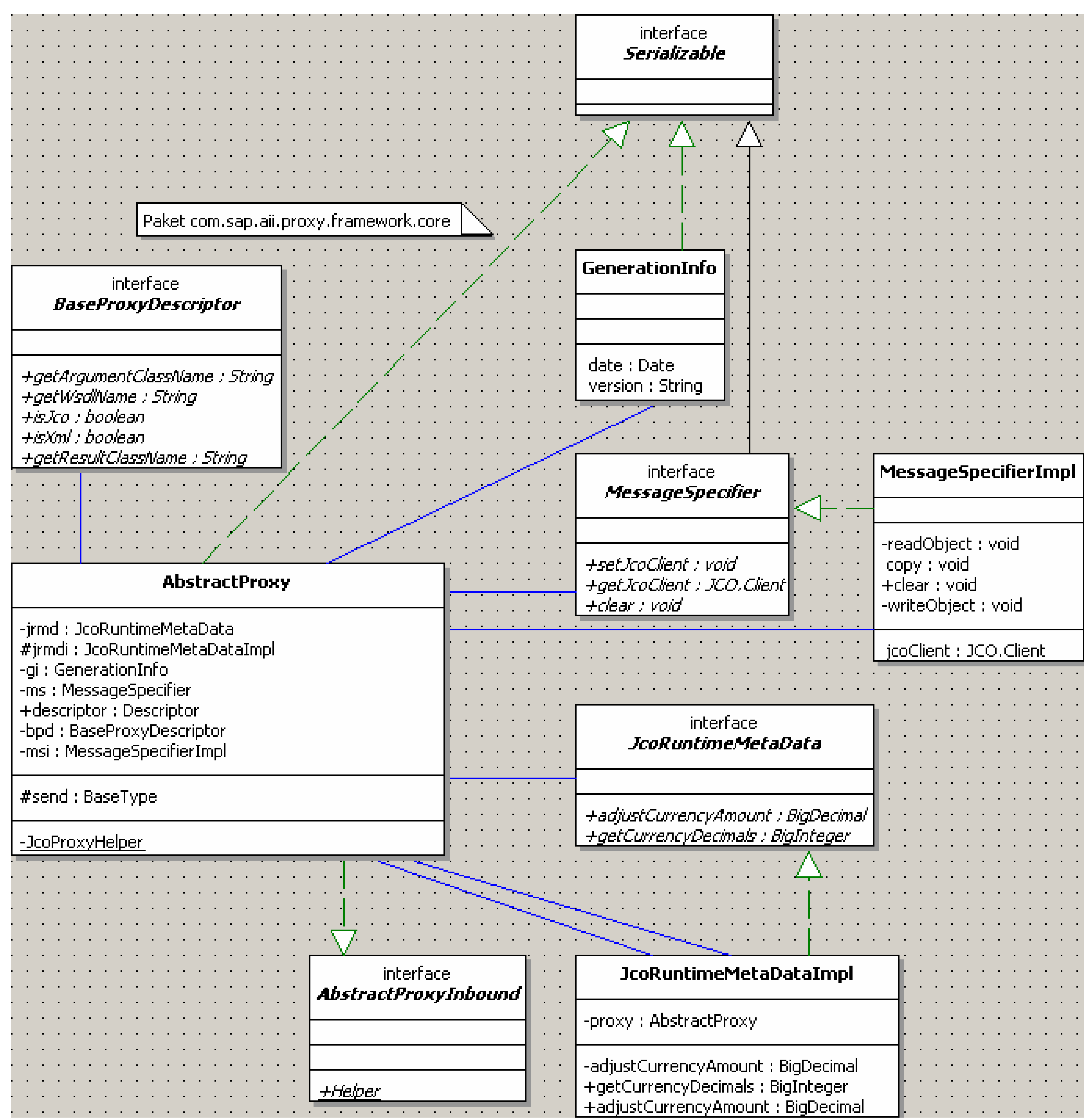


Abbildung 37: Stufe 2 der Modelklassenhierarchie bis LeadModel um die Klasse com.sap.aii.proxy.framework.core .AbstractProxy

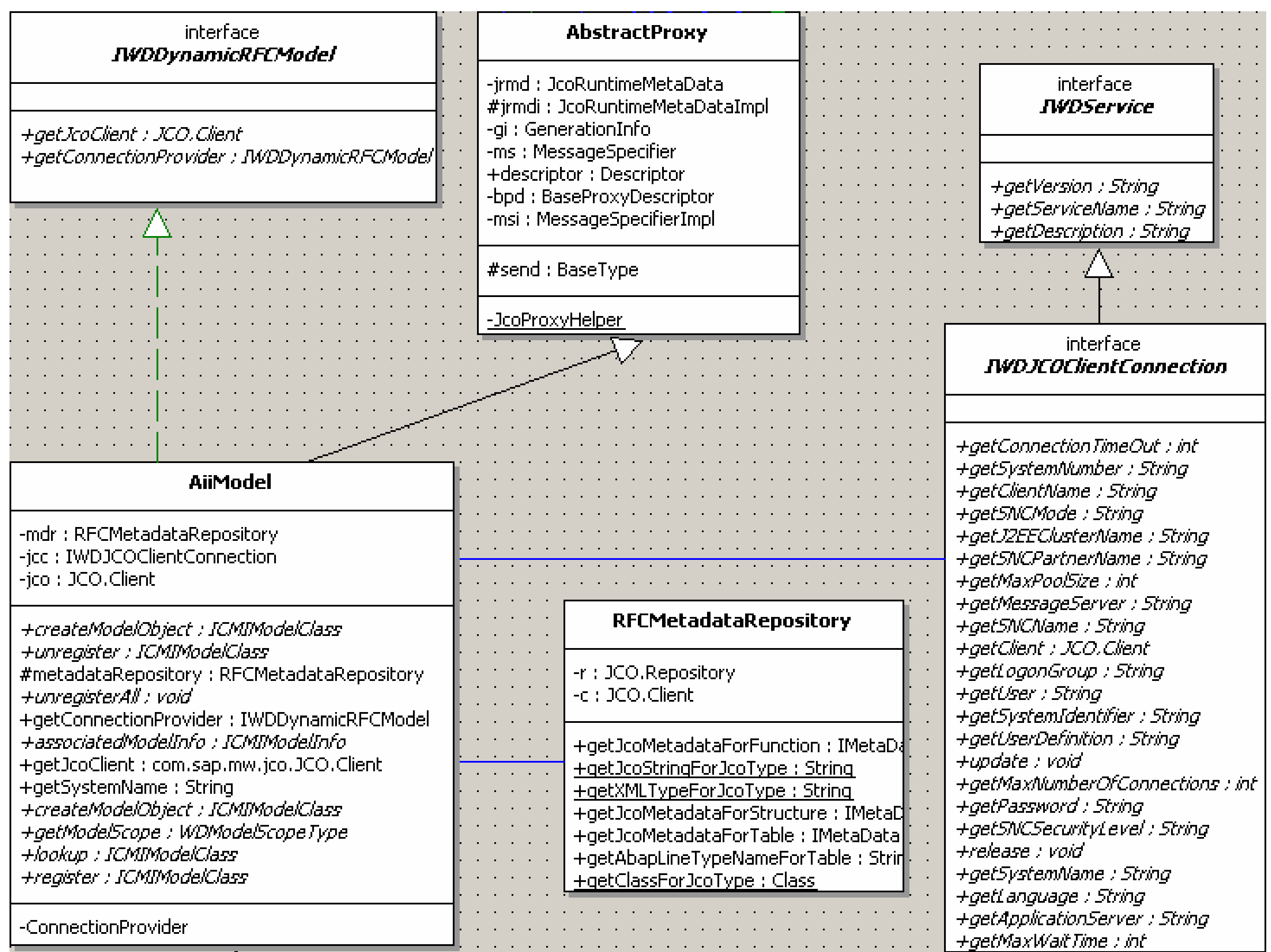


Abbildung 38: Stufe 3 der Modelklassenhierarchie bis LeadModel um die Klasse com.sap.tc.webdynpro.modelimpl.dynamicrfc.AiiModel

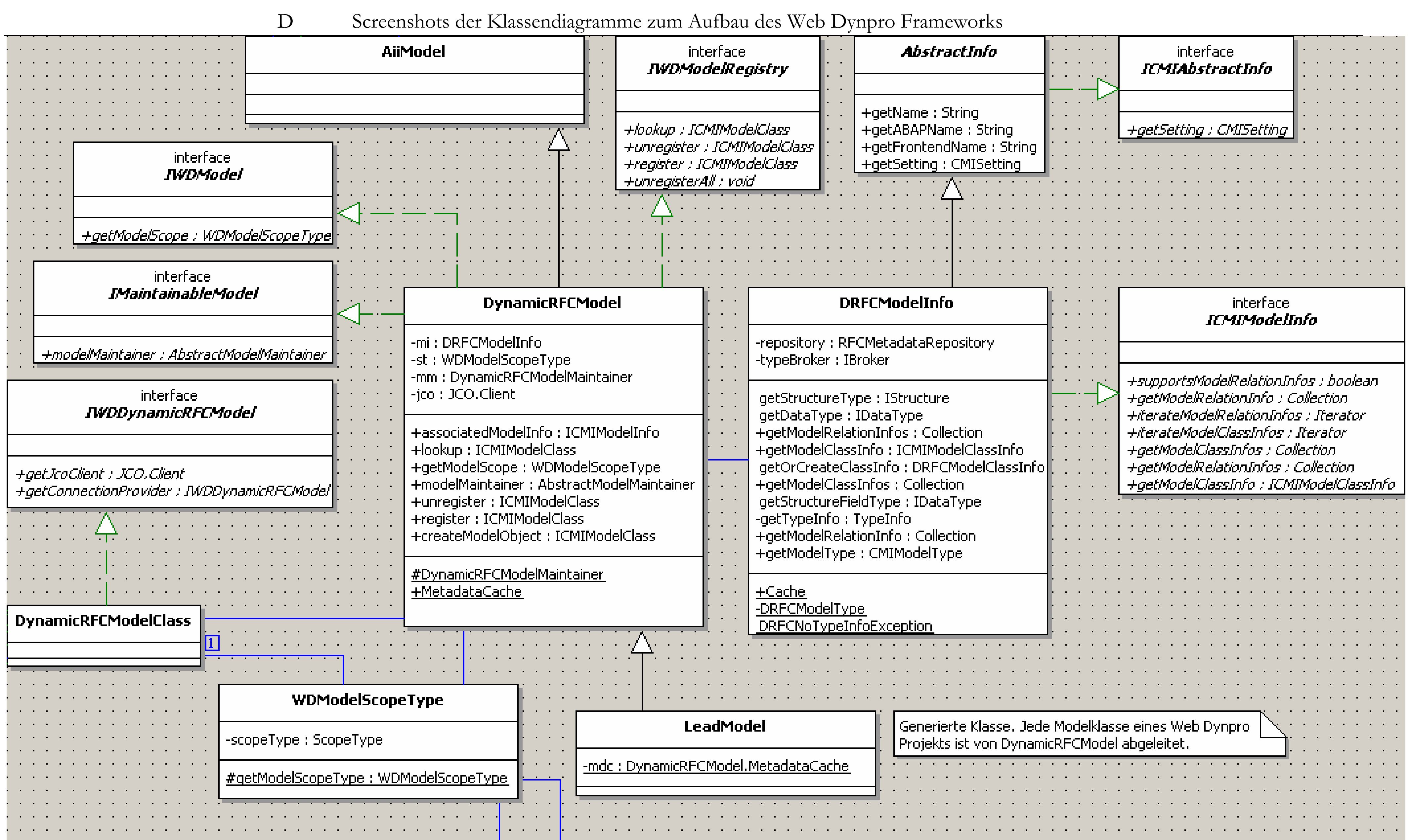


Abbildung 39: Stufe 4 und Ende der Modelklassenhierarchie mit LeadModel und der Klasse com.sap.tc.webdynpro.modelimpl.dynamicrfc.DynamicRFCModel mit der Metadatenklasse com.sap.tc.webdynpro.modelimpl.dynamicrfc.metadata.DRFCModelInfo

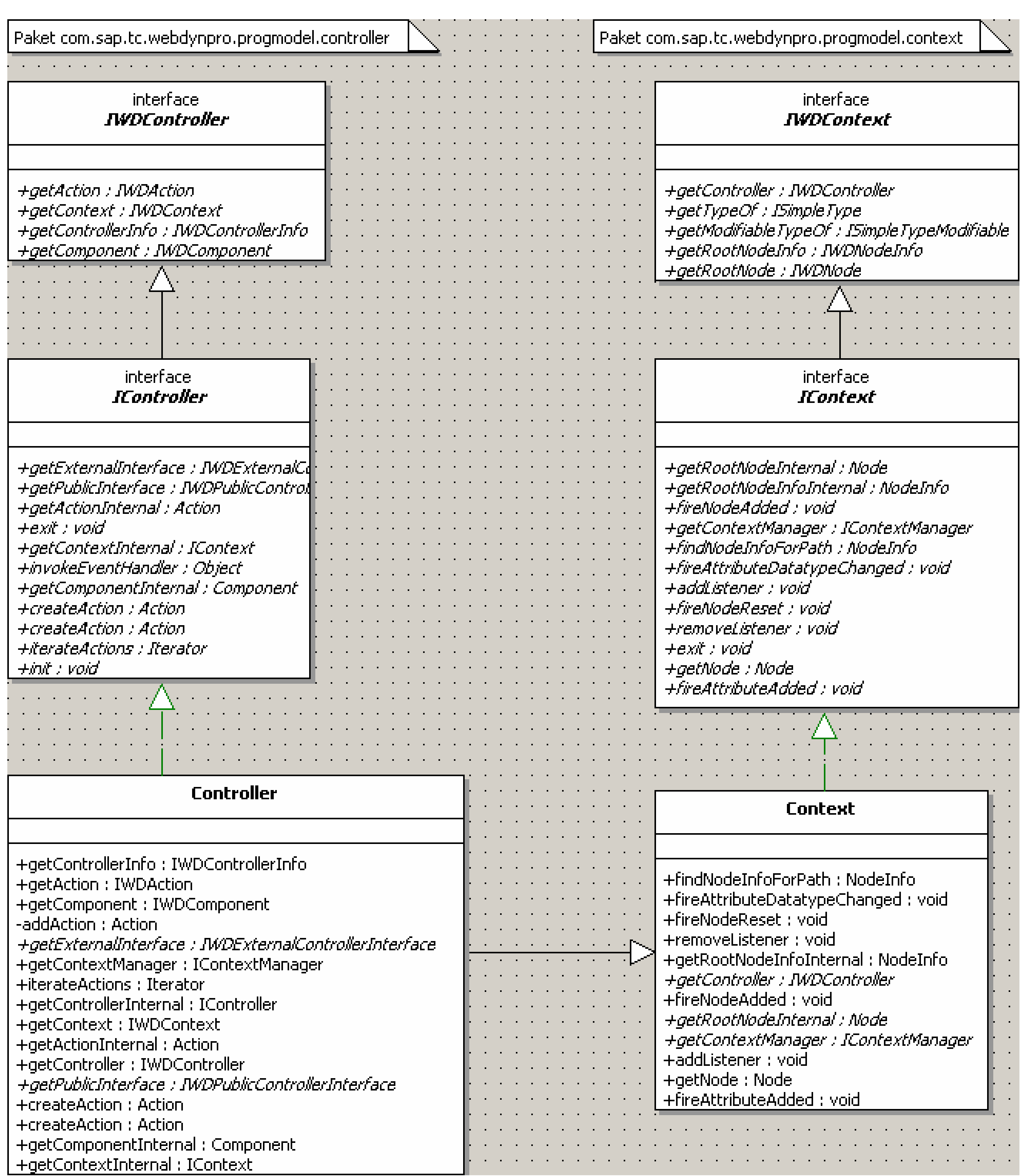


Abbildung 40: Basisinterfaces für Controller und Context und deren abstrakte Implementierungsklassen aus den Paketen com.sap.tc.webdynpro.progmodel.controller und com.sap.tc.webdynpro.progmodel.context

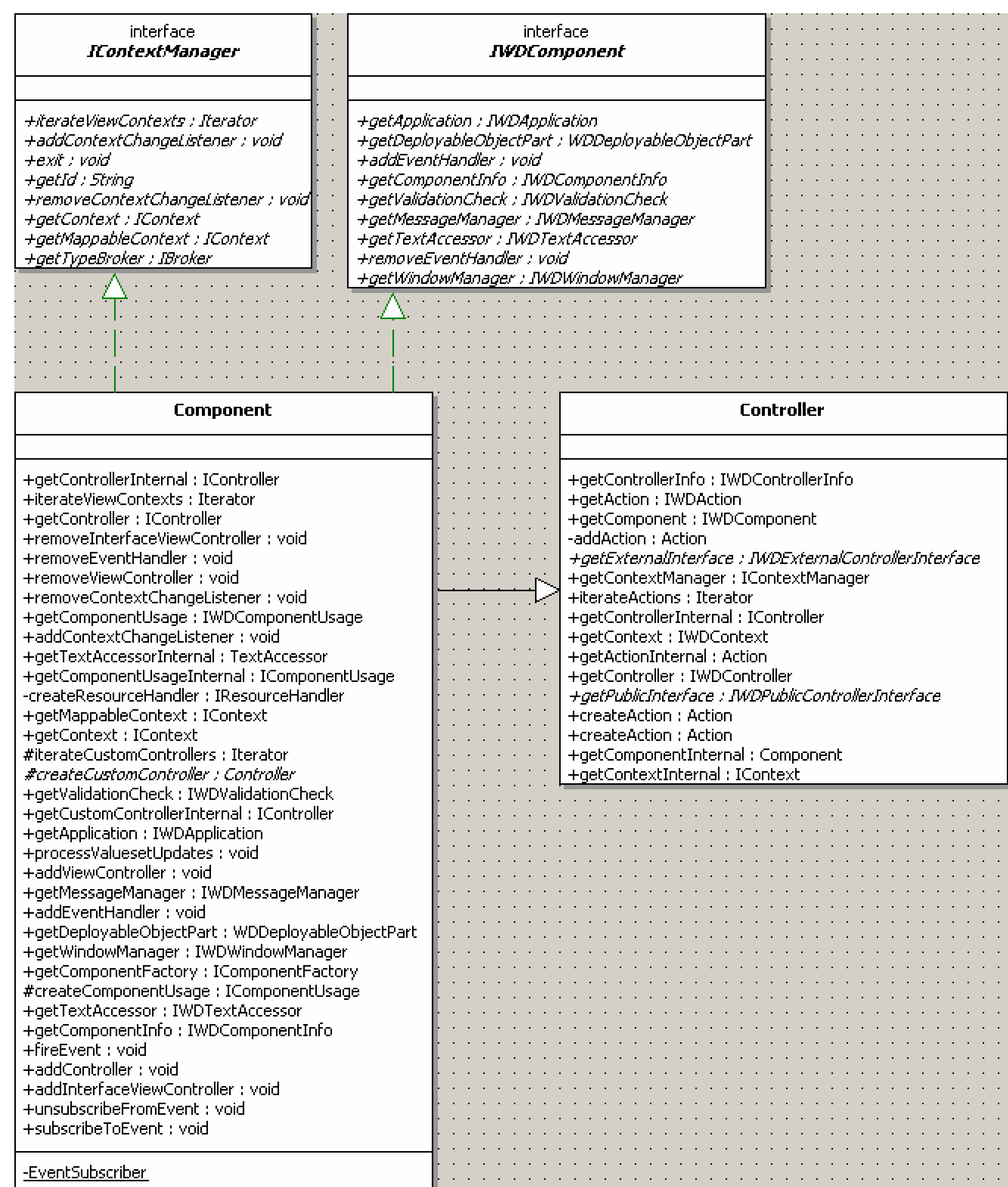


Abbildung 41: Interfaces und die abstrakte Implementierungsklasse Component. Zusammen mit Controller und Context bildet Component die Controller Komponente einer Web Dynpro Anwendung.

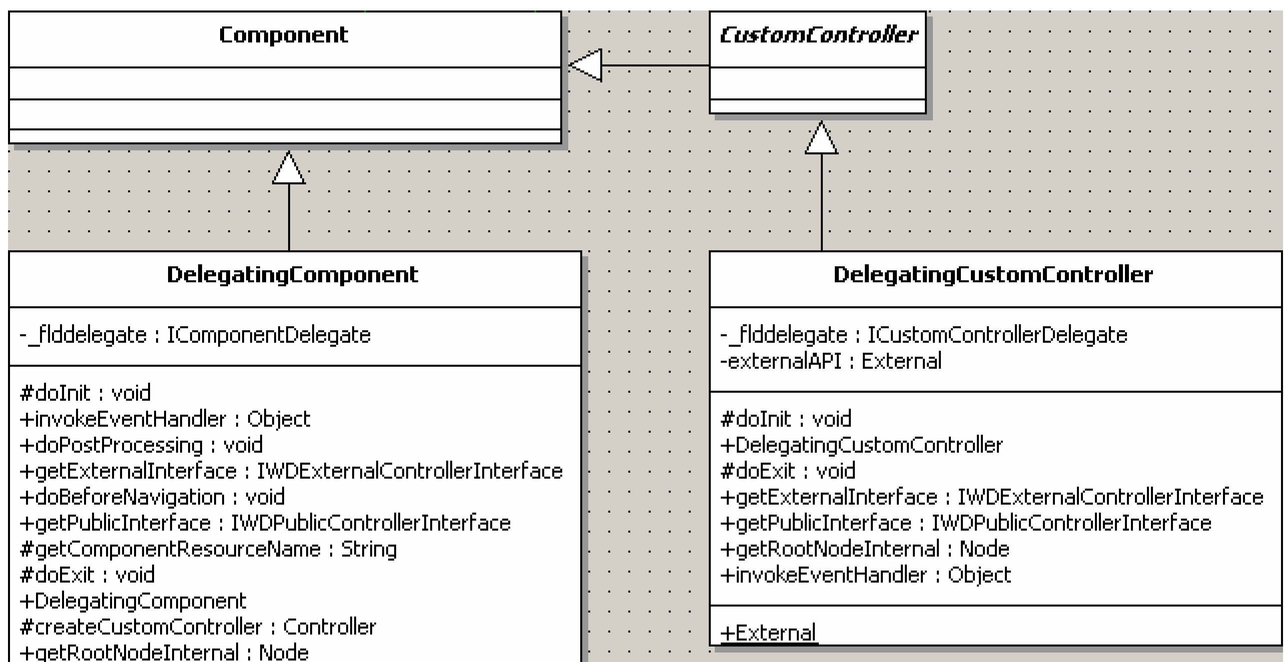


Abbildung 42: Derivate von Component und Controller als Schnittstellenklassen zu den generierten Klassen einer Controller Komponente.

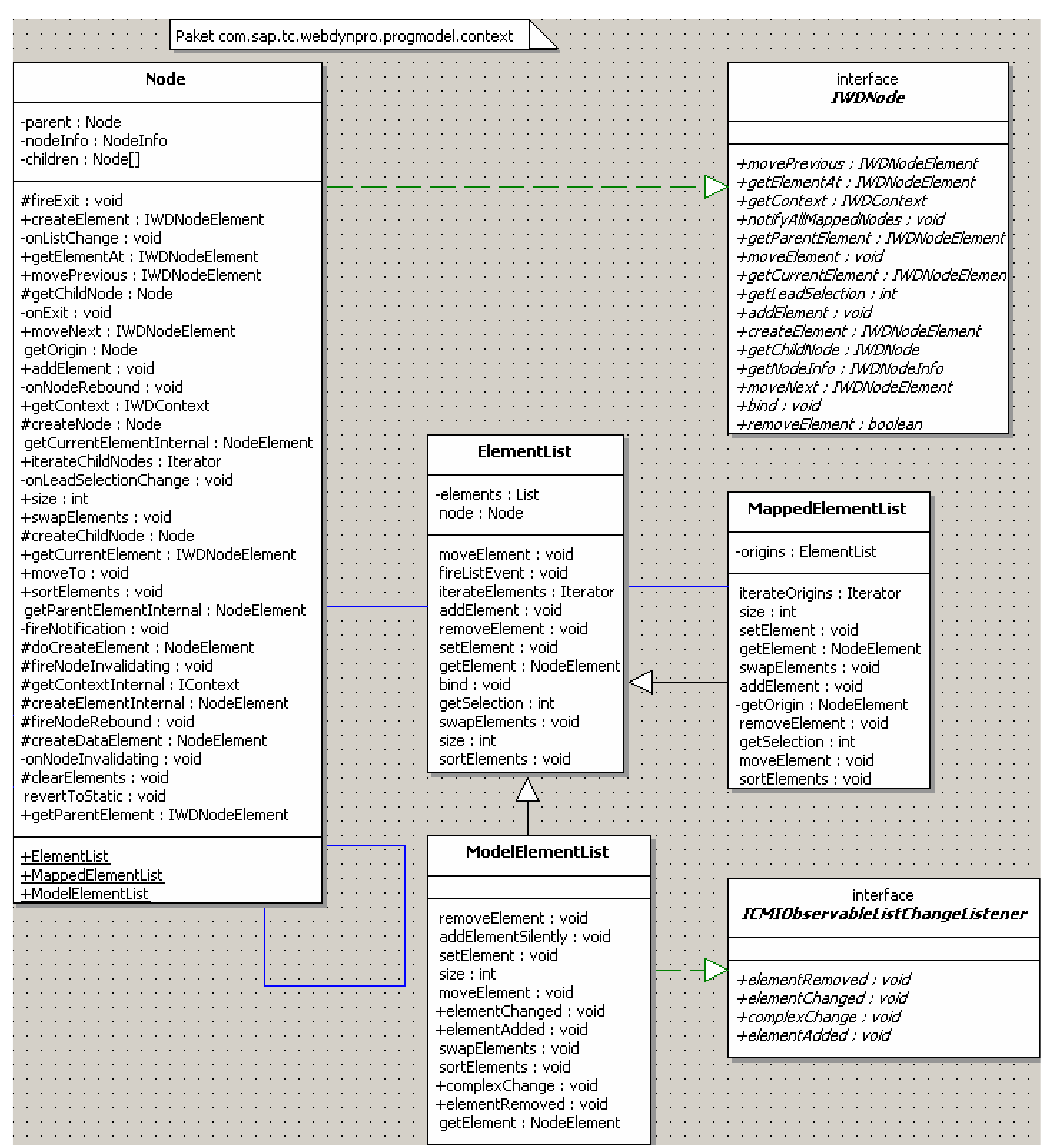


Abbildung 43: Basisinterface um die Klasse com.sap.tc.webdynpro.progmodel.context.Node und deren inner classes.

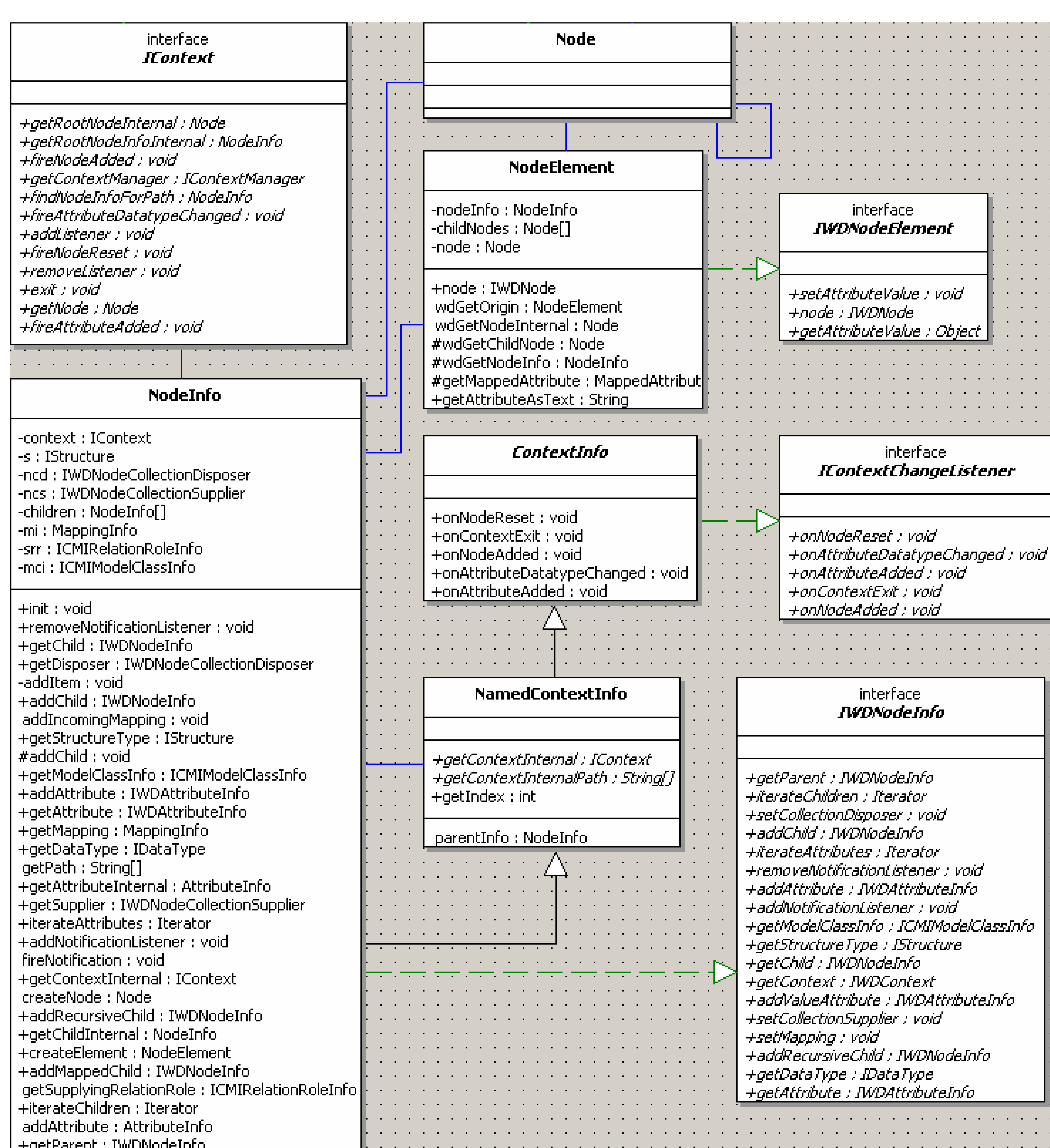


Abbildung 44: Die Node Metadatenklasse com.sap.tc.webdynpro.progmodel.context.NodeInfo und deren Basisklassen- und Interfaces.

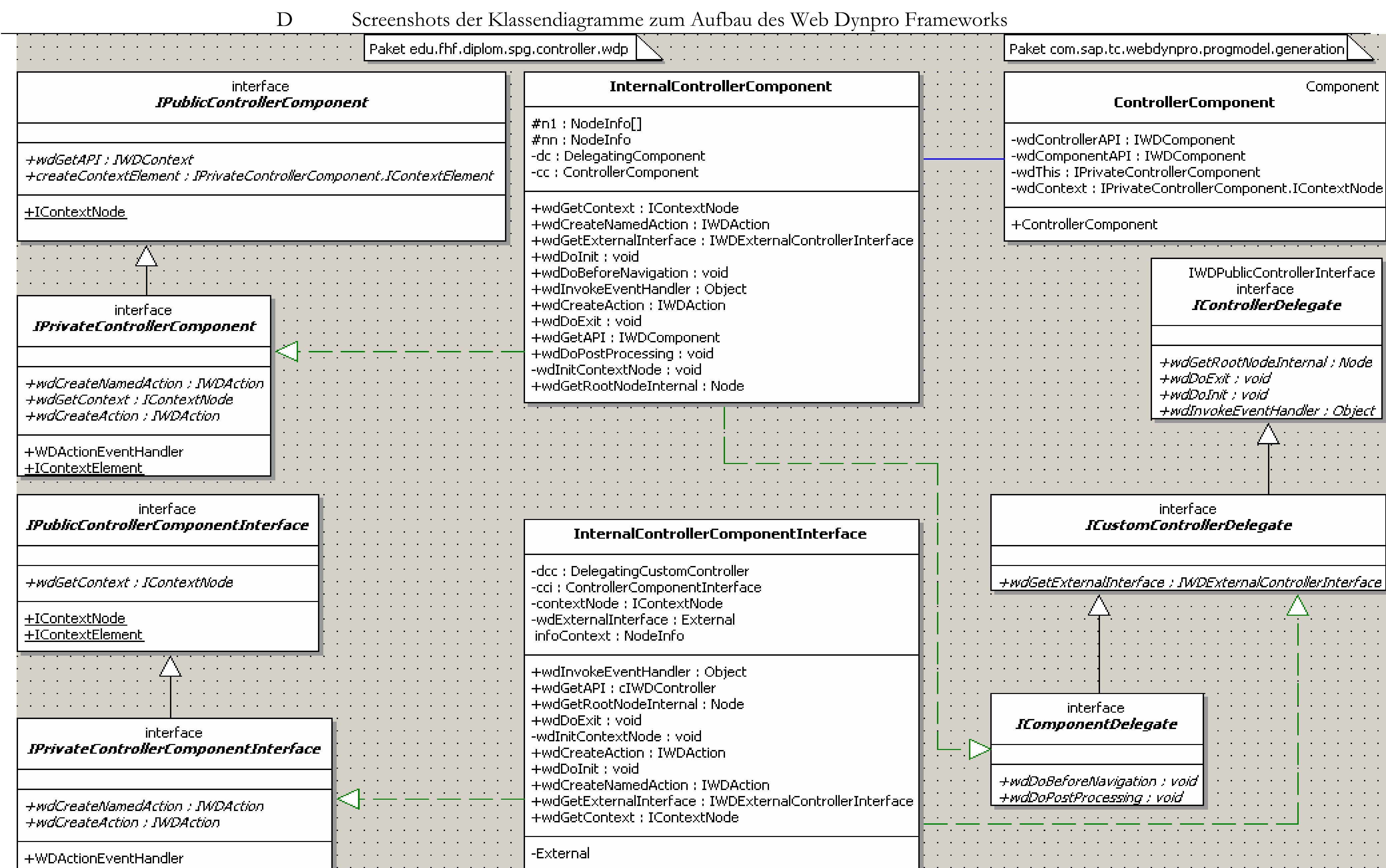


Abbildung 45: Die zentralen generierten Klassen InternalControllerComponent, InternalControllerComponentInterface und ControllerComponent für die Referenzanwendung

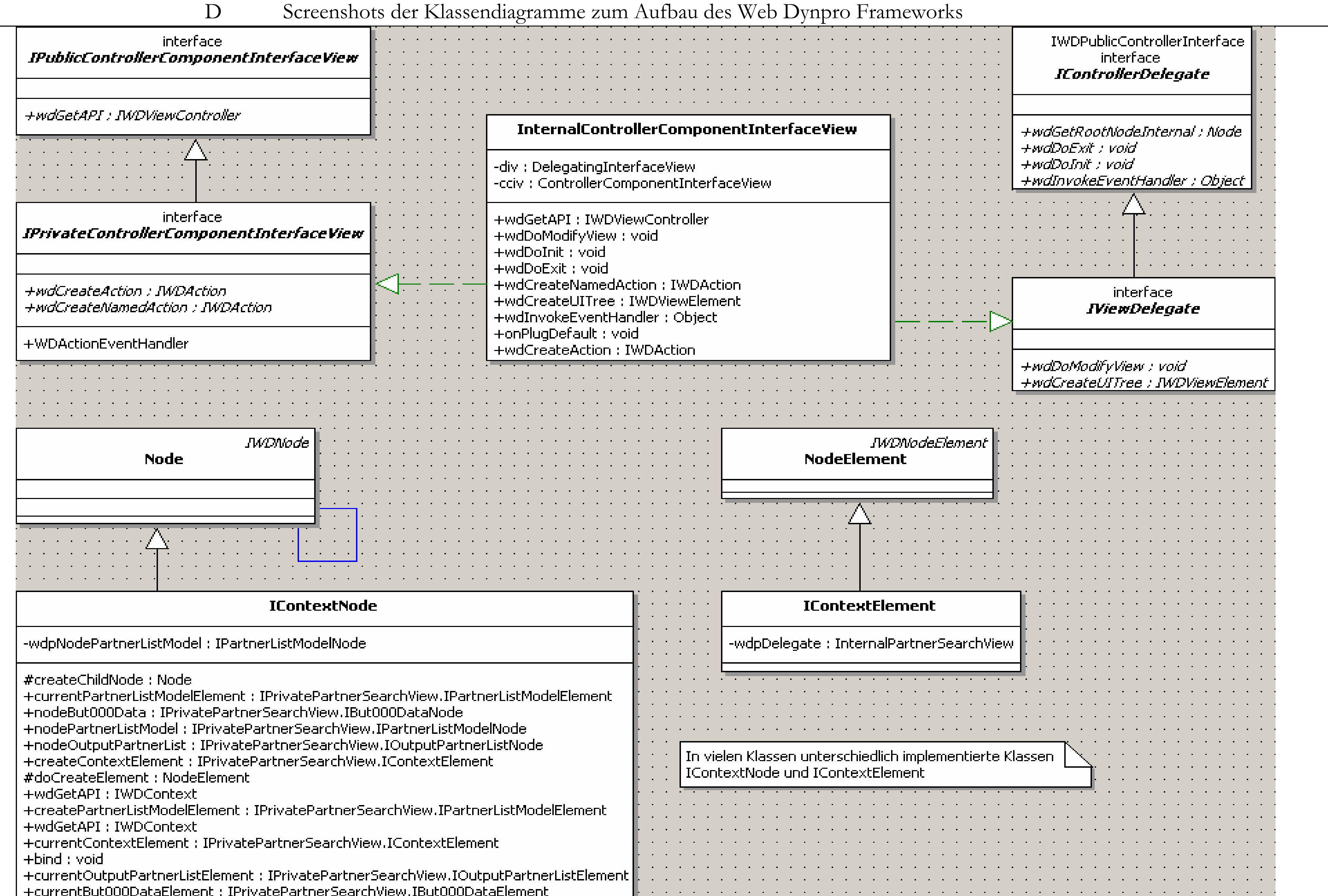


Abbildung 46: Die vierte zentrale generierte Klassen InternalControllerComponentInterfaceView und Node und NodeElement Derivate zur Abbildung von Context Strukturen

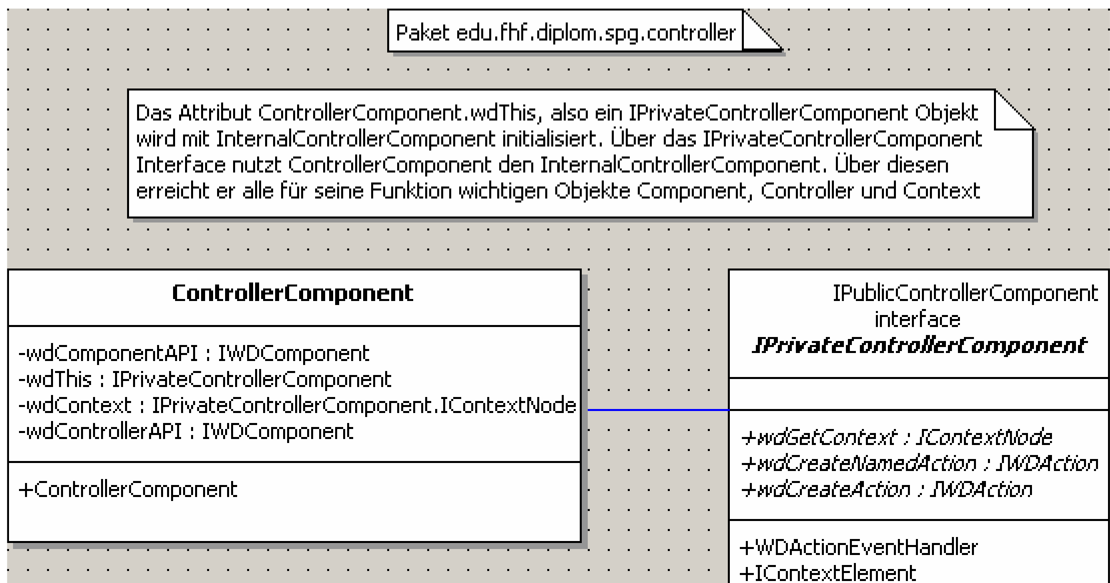


Abbildung 47: Die Klasse ControllerComponent als Controller Implementierungsklasse für den Entwickler.

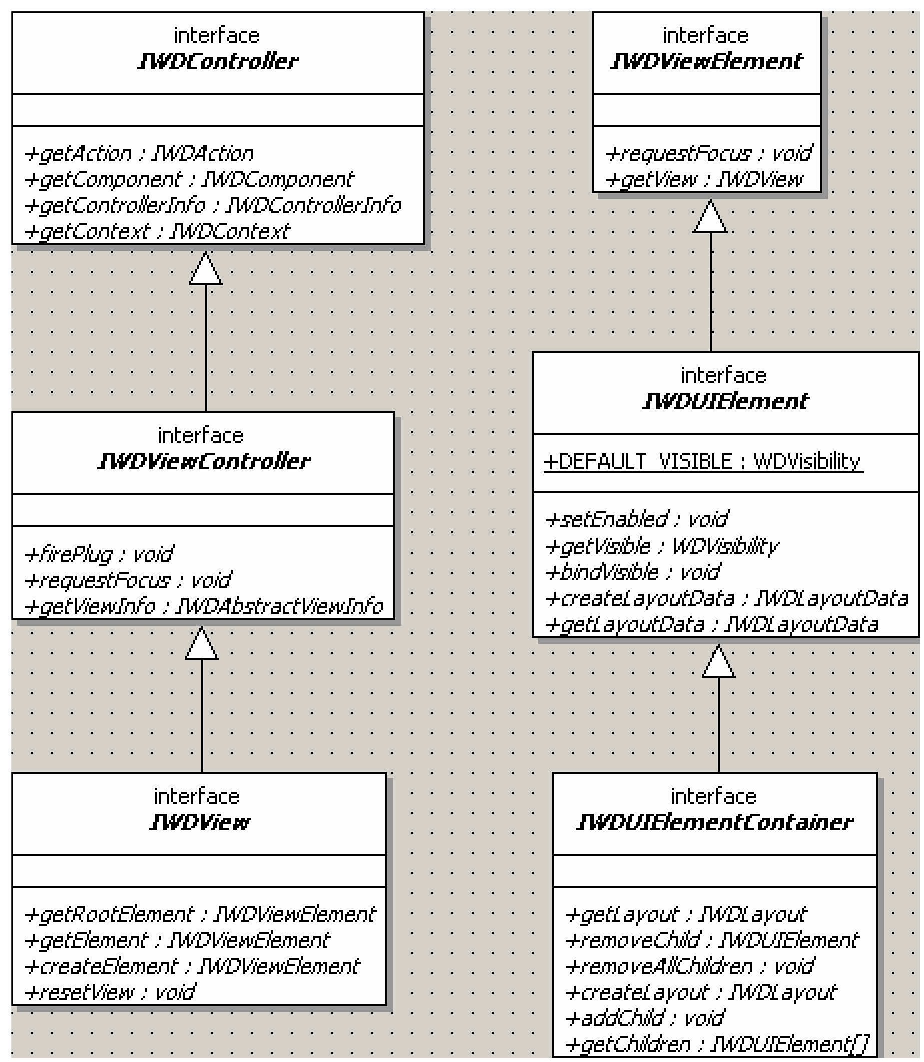


Abbildung 48: Das Basisinterface com.sap.tc.webdynpro.progmodel.api.IWDView und das Rootinterface IWDUIElement für alle Web Dynpro GUI Elemente, sowie IWDUIElementContainer für GUI Elemente, die andere IWDUIElement Objekte beinhalten können.

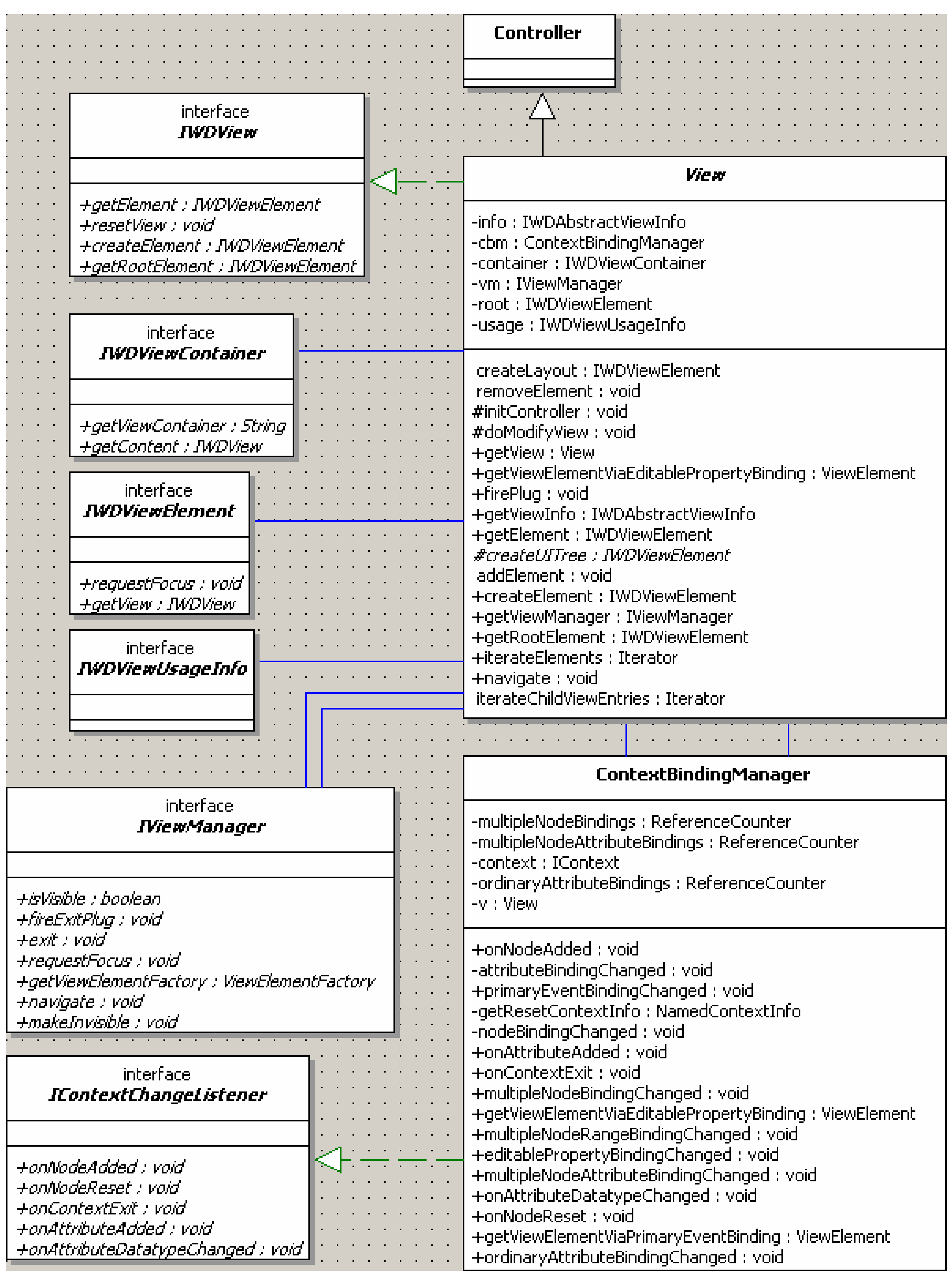


Abbildung 49: Die abstrakte Viewcomponentenbasisklasse com.sap.tc.webdynpro.progmodel.view.View und deren Umgebung. View ist von Controller abgeleitet, weil auch sie einen Context besitzt.

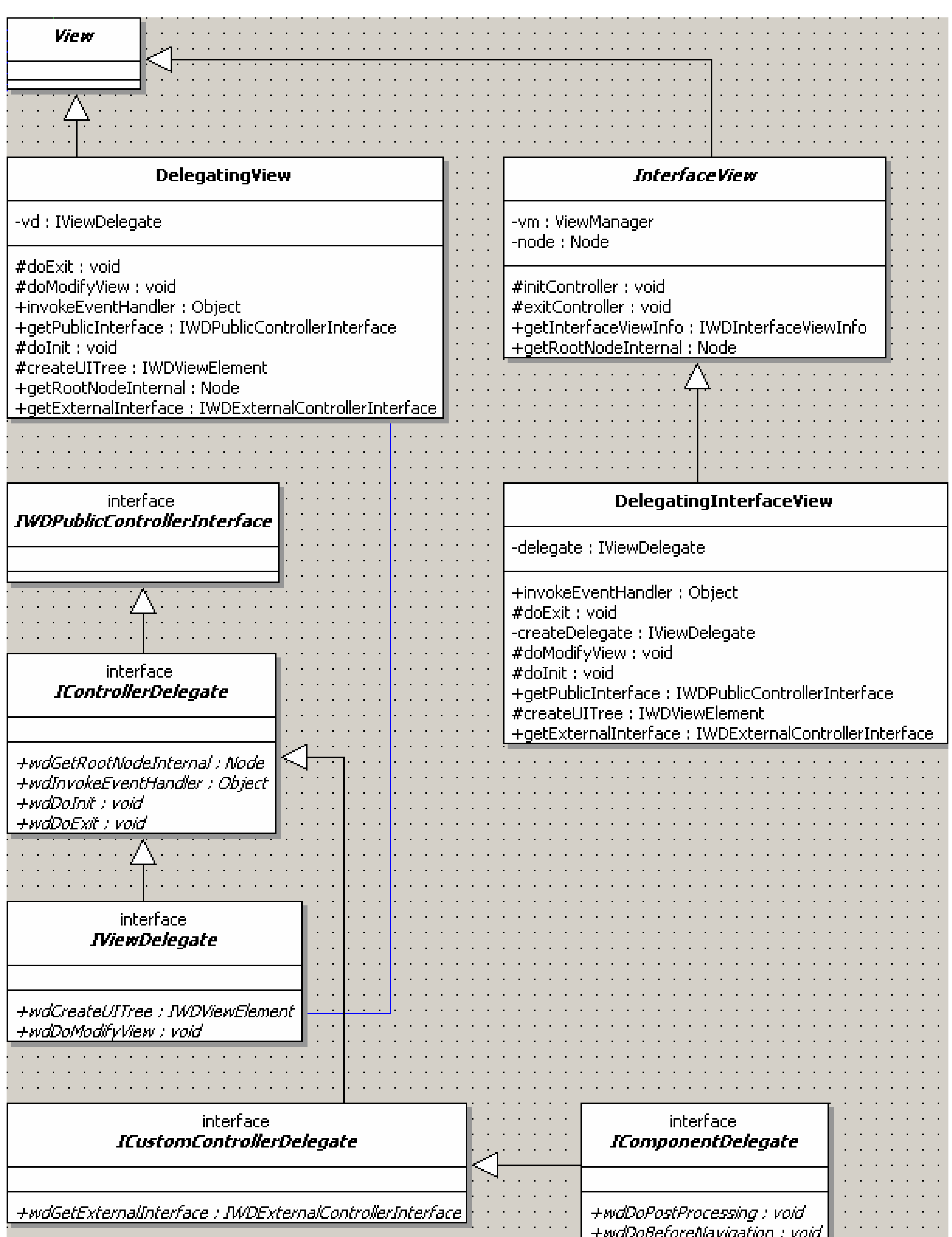


Abbildung 50: Derivate von View, DelegatingView, InterfaceView und DelegatingInterfaceView als Schnittstellenklassen zu den generierten Klassen einer View Komponente.

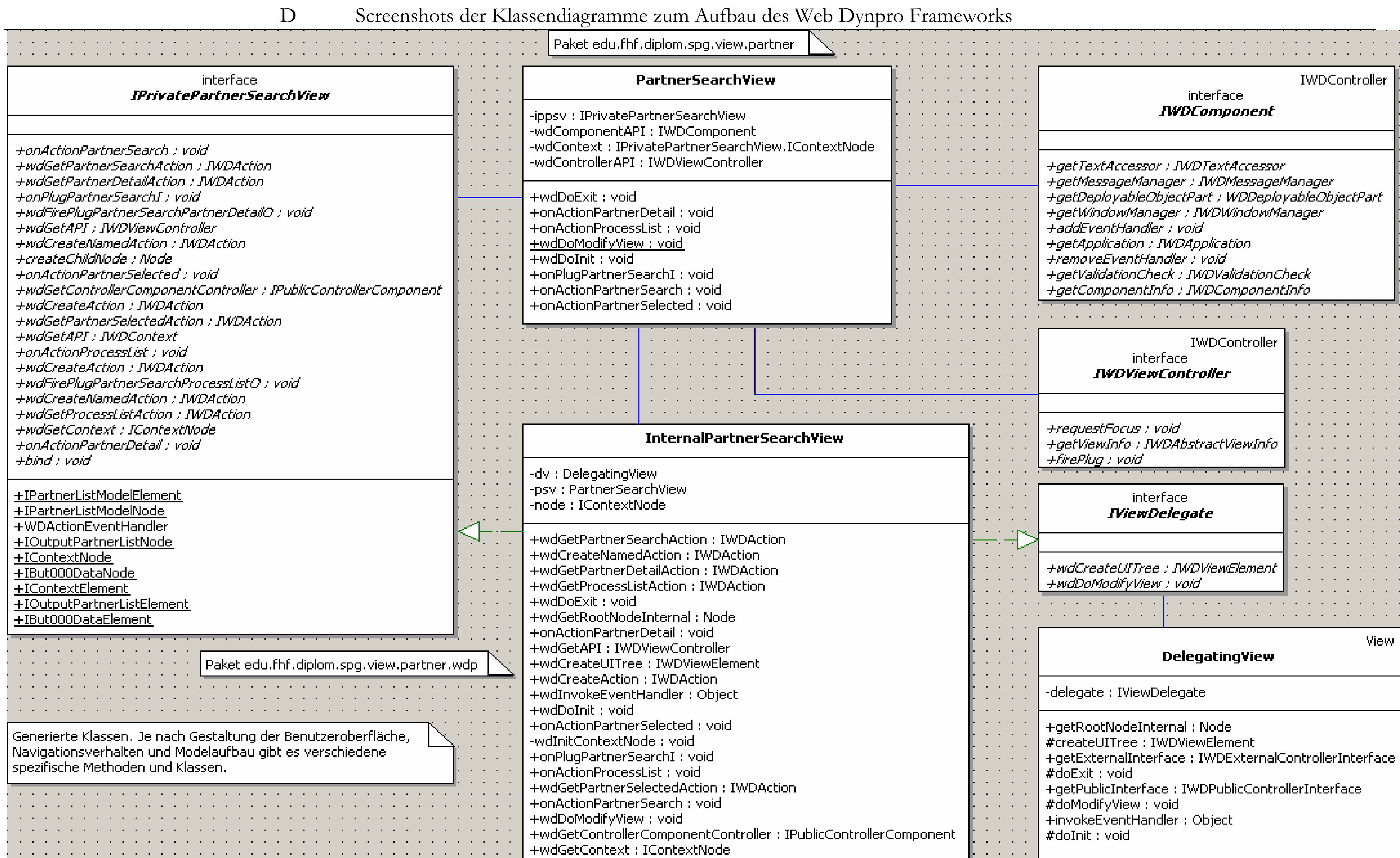


Abbildung 51: Die generierten Klasse einer Viewkomponente. Mit PartnerSearchView arbeitet der Entwickler. InternalPartnerSearchView beinhaltet das individuelle GUI Design der View.

6 Literaturverzeichnis

6.1 Internetressourcen

Struts Hauptseite
http://struts.apache.org/

Struts Downloadarchiv
http://archive.apache.org/dist/jakarta/

Struts 1.1 Source Code
http://archive.apache.org/dist/jakarta/struts/source/

Java Server Faces Hauptseite
http://java.sun.com/j2ee/javaserverfaces/index.jsp

Java Server Faces 1.1 Source Code
http://www.sun.com/software/communitysource/jsf/download.html

Web Dynpro Beipielrogramm Matrix
https://www.sdn.sap.com/sdn/developerareas/webdynpro.sdn?page=webdynpro_
tutorials.htm

Web Dynpro Hauptseite
https://www.sdn.sap.com/sdn/index.sdn
https://www.sdn.sap.com/sdn/developerareas/webdynpro.sdn?node=linkDnode6-2

SAP Download Area: Web Application Server
https://www.sdn.sap.com/sdn/downloadarea.sdn

6.2 Literatur

[1] Ted Husted, Cedric Dumoulin, George Franciscus, David Winterfeldt
Struts in Action
Building web applications with the leading Java framework
Manning Publications Co. 2003
ISBN: 1-930110-50-2

[2] Chuck Cavaness
Programming Jakarta Struts
O'Reilly Verlag
ISBN: 0-596-00328-5

[3] Craig McClanahan, Ed Burns, Roger Kitain
JavaServer™ Faces Specification
Version 1.1
Sun Microsystems
http://java.sun.com/

[4] Hans Bergsten
JavaServer Faces
O'Reilly Verlag April 2004
ISBN: 0-596-00539-3

[5] Austin Sincock
Enterprise Java for SAP
Apress™ Verlag
ISBN:1590590988

[6] Frédéric Heinemann, Christian Rau
SAP Web Application Server
Entwicklung von WebAnwendungen
SAP Press
ISBN 3-89842-213-5

[7] Rüdiger Buck-Emden, Peter Zencke
mySAP CRM
Kundenbezogene Geschäftsprozesse mit SAP CRM 4.0
SAP Press
ISBN 3-89842-380-8

[8] Michael Seubert, Thilo Krähmer
Geschäftspartner
Objektmodell CRM/EBP 4.0
Version 1.0 Mai 2003
Siehe auch Diplomarbeits CD in Verzeichnis Kapitel 3 - SAP CRM

[9] Michael Seubert, Dr. Jochen Rasch
Geschäftsvorgang
Objektmodell CRM/EBP 4.0
Version 1.0 Mai 2003
Siehe auch Diplomarbeits CD in Verzeichnis Kapitel 3 - SAP CRM

6.3 Abbildungsverzeichnis

Abbildung 1 Umsetzung des MVC Modells in Struts. Für Details siehe die folgenden Ausführungen
Abbildung 2 Servletgenerierung einer JSP mit <message> tag Ausschnitt.
Abbildung 3 Exception nach einer Benutzerinteraktion
Abbildung 4 Umsetzung des MVC Modells in JSF. Für Details siehe die folgenden Ausführungen
Abbildung 5 Der JSF Request Processing Lifecycle. Grundlegende Informationen zu jeder Phase werden direkt nachfolgend gegeben
Abbildung 6 Quelltext eines generierten Servelts für einen Command Button auf einer JSF. Die folgenden Erläuterungen stellen den Zusammenhang zwischen Tagattributen auf der JSP und den korrespondierenden Methodenaufrufen der Tag Repräsentationsklassen dar.
Abbildung 7 Ausschnittt aus der standard-html-renderkit.xml für eine HTML Submit Schaltfläche. Was notwendig ist, damit alle HTML GUI Elemente für das JSF Framework verfügbar sind, wird nachfolgend dargestellt.
Abbildung 8 Codelisting über das Starten des JSF Request Processing Lifecycle durch den Tomcat Webserver, der Ausgangspunkt für alle direkt folgenden Untersuchungen.
Abbildung 9 JSP Codeausschnitt mit Name und Passwort Textfeldern und Submit Schaltfläche.
Abbildung 10 Zugehöriger faces-config.xml Abschnitt für die JSP aus Abbildung 8 und das Beanobjekt als Modelklasse.
Abbildung 11 Einordnung von betriebswirtschaftlichen Sachgebieten in die vier Geschäftsobjekte SAP CRM Architektur
Abbildung 12 Zuordnung einer Hierarchiegruppe zu einem Geschäftspartner
Abbildung 13 Zuordnung einer Geschäftspartnerbeziehung zu einem Geschäftspartner
Abbildung 14 Umgebungstabellen des Geschäftspartners mit ausgelagerten Daten zur Verbindung mit den Customizing Tabellen.
Abbildung 15 Verbindung von Adressdatenentitäten zur Geschäfstpartnerhaupttabelle BUT000
Abbildung 16 SAP CRM Customizing Tabellen des Geschäftspartners
Abbildung 17 Beziehungskonstrukt zwischen Geschäftspartnerhaupttabelle und Geschäftspartnerrollen
Abbildung 18 Zusammenhang von Geschäftspartnerbeziehungsart und Geschäftspartnerrolle
Abbildung 19 Die Ansprechpartnerbeziehung eines Geschäftspartners
Abbildung 20 Geschäftspartnerattribute
Abbildung 21 Das Entity Relationship Modell des SAP CRM Geschäftsvorgangs
Abbildung 22 Beziehung der Tabellen CRMD_CANCEL und CRMD_CANCEL_IR
Abbildung 23 Das Entity Relationship Modell der Customizing Tabellen des Geschäftsvorgangs
Abbildung 24 Darstellung der Weiterleitung von Ereignissen durch Änderungen an Komponenten an andere Komponenten durch den Event Handler
Abbildung 25 Ein Web Dynpro Projekt im Über-blick in der Eclipse Perspektive des Web Dynpro Explorers
Abbildung 26 Generierte xml Dateien für die Datenfelder eines ABAP Funktionsbausteins in einem Web Dynpro Projekt
Abbildung 27 Übersicht über eine Modelkomponente im Web Dynpro Explorer
Abbildung 28 Der Controller Kontext einer Controller Komponente eines Web Dynpro Projekts
Abbildung 29 Der Navigation Modeler des SAP Developer Studios zur Modellierung des Navigationsverhaltens zwischen Benutzeroberflächen
Abbildung 30 Detailliertere Ansicht eines Modelobjekts im Kontext
Abbildung 31 Basisinterfaces für CMI Modelklassen aus dem Paket com.sap.tc.cmi.model
Abbildung 32 Basisinterfaces aus dem Paket com.sap.tc.cmi.metadata für Klassen, die Metadaten über Modelklassen liefern
Abbildung 33 Stufe 1 der Modelklassenhierarchie bis Z_All_Leads_By_Partner_2_Input um die Klasse com.sap.aii.proxy.framework.core.AbstractType
Abbildung 34 Stufe 2 und 3 der Modelklassenhierarchie bis Z_All_Leads_By_Partner_2_Input um die Klassen AiiModelClass und DynamicRFCModelClass
Abbildung 35 Stufe 4 der Modelklassenhierarchie bis Z_All_Leads_By_Partner_2_Input um die Klasse DynamicRFCModelClassExecutable
Abbildung 36 Basisinterfaces für ICMIModel Derivate aus dem Paket com.sap.tc.webdynpro.progmodel.model.api
Abbildung 37 Stufe 1 der Modelklassenhierarchie bis LeadModel um die Klasse com.sap.aii.proxy.framework.core.AbstractProxy
Abbildung 38 Stufe 2 der Modelklassenhierarchie bis LeadModel um die Klasse com.sap.tc.webdynpro.modelimpl.dynamicrfc.AiiModel

Abbildung 39 Stufe 3 und Ende der Modelklassenhierarchie mit LeadModel und der Klasse com.sap.tc.webdynpro.modelimpl.dynamicrfc.DynamicRFCModel mit der Metadatenklasse com.sap.tc.webdynpro.modelimpl.dynamicrfc.metadata.DRFCModelInfo

Abbildung 40 Basisinterfaces für Controller und Context und deren abstrakte Implementierungsklassen aus den Paketen com.sap.tc.webdynpro.progmodel.controller und com.sap.tc.webdynpro.progmodel.context

Abbildung 41 Interfaces und die abstrakte Implementierungsklasse Component. Zusammen mit Controller und Context bildet Component die Controller Komponente einer Web Dynpro Anwendung.

Abbildung 42 Derivate von Component und Controller als Schnittstellenklassen zu den generierten Klassen einer Controller Komponente.

Abbildung 43 Basisinterface um die Klasse com.sap.tc.webdynpro.progmodel.context.Node und deren inner classes.

Abbildung 44 Die Node Metadatenklasse com.sap.tc.webdynpro.progmodel.context.NodeInfo und deren Basisklassen- und Interfaces.

Abbildung 45 Die zentralen generierten Klassen InternalControllerComponent, InternalController ComponentInterface und ControllerComponent für die Referenzanwendung

Abbildung 46 Die vierte zentrale generierte Klassen InternalControllerComponentInterfaceView und Node und NodeElement Derivate zur Abbildung von Context Strukturen

Abbildung 47 Die Klasse ControllerComponent als Controller Implementierungsklasse für den Entwickler.

Abbildung 48 Das Basisinterface com.sap.tc.webdynpro.progmodel.api.IWDView und das Rootinterface IWDUIElement für alle Web Dynpro GUI Elemente, sowie IWDUIElementContainer für GUI Elemente, die andere IWDUIElement Objekte beinhalten können.

Abbildung 49 Die abstrakte Viewcomponentenbasisklasse com.sap.tc.webdynpro.progmodel.view.View und deren Umgebung. View ist von Controller abgeleitet, weil auch sie einen Context besitzt.

Abbildung 50 Derivate von View, DelegatingView, InterfaceView und DelegatingInterfaceView als Schnittstellenklassen zu den generierten Klassen einer View Komponente.

Abbildung 51 Die generierten Klasse einer Viewkomponente. Mit PartnerSearchVIew arbeitet der Entwickler. InternalPartnerSearchView beinhaltet das individuelle GUI Design der View.

Abbildung 52 Der Kontext der Viewkomponente PartnerSearchView

6.4 Tabellenverzeichnis

Tabelle 1 Attribute des <message> Tag aus den Struts Tag Libraries
Tabelle 2 Die Bedeutung der Servlet API Objekte HttpSession, ServletContext und ServletConfig
Tabelle 3 Wichtige web.xml Attribute zur Konfiguration eines ActionServlets
Tabelle 4 JSF Implementierungshersteller und freie JSF GUI Komponenten
Tabelle 5 Die vier Sichten auf die Geschäftsobjekte des SAP CRM 4.0. Diese bilden die grundlegende Strukturierung der Hauptentitäten im SAP CRM.
Tabelle 6 Tabellentypen SAP CRM. Jede Tabelle eines SAP CRM Geschäftsobjekts gehört einem dieser Typen an.
Tabelle 7 Generalisierungstypen von SAP CRM
Tabelle 8 SAP CRM Extension Tabellen des Geschäftspartners
Tabelle 9 Extensiontabellen des Geschäftsvorgangs
Tabelle 10 Felder der Tabelle CRMD_FINPROD_I
Tabelle 11 Felder der Tabelle BBP_PDACC
Tabelle 12 Felder der Tabelle BBP_PDATT
Tabelle 13 Felder der Tabelle CRMD_BILLING
Tabelle 14 Felder der Tabelle BBP_PDACC
Tabelle 15 Felder der Tabelle CRMD_PRICING
Tabelle 16 Felder der Tabelle CRMD_QUALIF
Tabelle 17 Felder der Tabelle BBP_PDTAX
Tabelle 18 Felder der Tabelle CRMD_SALES
Tabelle 19 Felder der Tabelle CRMD_SHIPPING
Tabelle 20 Felder der Tabelle BBP_PDLIM
Tabelle 21 Felder der Tabelle SCAPPTSEG
Tabelle 22 Die Java Archive aus denen Web Dynpro besteht