Inhaltsverzeichnis

1 Die zen Platform.....7

1.1 Die Entwicklungsumgebung.....7

1.2 Das Laufzeitsystem.....7

2 Überblick über die zen Engine.....8

2.1 Architektur der zen Engine.....8

2.2 Funktionsweise.....8

2.3 Services.....9

3 Aufbau einer zen-Anwendung.....10

3.1 Workflowmodell.....10

3.2 Datenmodell.....11

3.3 Verknüpfung von Workflow und Daten.....11

3.4 Geschäftslogik.....12

3.4.1 Workflow Operations.....12

3.4.2 Business Rules.....13

3.5 Ressourcen und Attribute.....13

4 Ablauf einer zen-Anwendung.....14

4.1 Grundlagen.....14

4.2 Überblick.....15

4.3 Validierung und Datenübernahme.....16

4.3.1 Workflow Validation (Level 1).....16

4.3.2 Request Validation (Level 2 + 3).....17

4.3.3 Verschmelzung von Request- und Sessiondaten.....17

4.3.4 Domain Validation (Level 4).....18

4.3.5 Computation Rules.....18

4.3.6 Validation Rules (Level 5).....18

4.4 Workflowsteuerung.....19

4.5 Sessionübernahme.....20

4.6 Datenausgabe.....20

5 Anwendungsentwicklung mit dem zen Developer.....22

5.1 Erstellen einer neuen Anwendung.....23

5.1.1 Anlegen eines neuen XSL Stylesheets.....23

5.1.2 Application View.....24

5.2 Verwalten des Repository.....24

5.3 Öffnen und Bearbeiten einer Anwendung.....24

5.4 Erstellen eines Workflows.....25

5.4.1 Workflow View.....26

5.4.2 Action View.....27

5.5 Erstellen eines Datenmodells.....28

5.5.1 Data Model View.....28

5.5.2 Datatype View.....30

5.5.3 Domain View....30
5.6 Erstellen der Geschäftslogik....32
5.7 Erstellen von Ressourcen....34
5.8 Debugging einer Anwendung....36
6 zen Engine Referenz....37
6.1 Service-API....37
6.1.1 Connection....37
6.1.2 JDO....38
6.1.3 Messaging....38
6.1.4 Transaction....39
6.1.5 Logging....40
6.1.6 Mail....40
6.1.7 SessionManagement....41
6.1.8 ResourceRepository....41
6.1.9 ComponentSelector....42
6.2 FOM API....42
6.2.1 Element....43
6.2.2 Atom....43
6.2.3 ElementCollection....44
6.2.4 Composition....44
6.2.5 List....44
6.2.6 ElementBuilder....45
6.2.7 Path....45
6.3 Core API....46
6.3.1 OperationException....46
6.3.2 OperationRuntimeException....47
6.3.3 AggregationOperation....47
6.3.4 Domain....47
6.3.5 DataConversion....48
6.4 Core-Funktionen....48
6.4.1 Feldfunktionen....48
6.4.2 Spezielle Attribute....48
6.5 Hook API....49
6.5.1 RequestModificator....49
6.5.2 Interceptor....51
6.5.3 StylesheetSelector....52
6.6 Schnittstellen....52
6.6.1 Eingabeformate....52
6.6.2 Ausgabeformat....55
6.7 Frontends....57
6.7.1 GenericServlet....57
6.7.2 StatefulBaseServlet, StatelessBaseServlet....58

6.7.3 BaseKernelClient....58
6.8 XSL-Stylesheets....59
6.8.1 generic.xsl....59
6.8.2 components.xsl....60
6.8.3 xml.xsl....61
7 Anwendungsdesign....62
7.1 Basis....62
7.1.1 Technische Beschränkungen....62
7.1.2 Workflowmodellierung....62
7.1.3 Datenmodellierung....63
7.1.4 Operations....64
7.1.5 Domänen....67
7.1.6 Datentypen....68
7.1.7 Stylesheets....68
7.2 Patterns....69
7.2.1 Ein- und Ausgabe mit In-Opt und Out-Opt....69
7.2.2 Data Grids in HTML....70
7.2.3 Business Rules auf programmatisch gefüllten Eingabefeldern....71
7.2.4 Listen mit Blätterlogik....72
7.2.5 Ausblendbare Buttons....72
7.2.6 Asynchroner Prozeß / Managed Message....73
8 Konfiguration der zen Platform....76
8.1 Formate: Deployment-Konfiguration....76
8.1.1 FileBasedDeploymentConfigurationFactory....76
8.1.2 JNDIBasedDeploymentConfigurationFactory....76
8.2 Referenz: Deployment-Konfiguration....76
8.2.1 Deployment-Varianten....77
8.2.2 Basis-Konfiguration....77
8.2.3 Spezialisierung für Single Container Deployments und das Frontend bei Cluster Deployments....79
8.2.4 Spezialisierung für Backends eines Cluster Deployments....79
8.2.5 Monitoring und Journaling von Components....80
8.3 Formate: Service-Konfiguration....81
8.3.1 FileBasedServiceConnectorFactory....81
8.3.2 JNDIBasedServiceConnectorFactory....81
8.4 Referenz: Service-Konfiguration....81
8.4.1 Logging-Service....82
8.4.2 Mail-Service....87
8.4.3 Transaction-Service....87
8.4.4 Connection-Service....88
8.4.5 JDO-Service....89
8.4.6 Messaging-Service....90

8.4.7 ResourceRepository-Service....91
8.4.8 SessionManager-Service....92
8.4.9 ComponentSelector-Service....92
8.4.10 SCFManagement-Service....92
8.5 Start der zen-Engine....93
8.5.1 Klassenbibliotheken....93
8.5.2 Startparameter....94
A Konfiguration Deployment....96
A.1 DTD scfdeploy.xml....96
A.1.1 Beispielkonfiguration: Single Container Deployment....96
A.1.2 Beispielkonfiguration für Cluster Deployment....97
A.1.3 Beispielkonfiguration für Monitoring & Journaling....99
B Konfiguration Service....99
B.1 DTD scfservice.xml....99
B.1.1 Beispielkonfiguration Logging-Service....100
B.1.2 Beispielkonfiguration Mail-Service....100
B.1.3 Beispielkonfiguration Transaction-Service....100
B.1.4 Beispielkonfiguration Connection-Service....101
B.1.5 Beispielkonfiguration JDO-Service....101
B.1.6 Beispielkonfiguration Messaging-Service....101
B.1.7 Beispielkonfiguration ResourceRepository-Service....101
B.1.8 Beispielkonfiguration SessionManager-Service....102
B.1.9 Beispielkonfiguration ComponentSelector-Service....102
B.1.10 Beispielkonfiguration SCFManagement-Service....102

Vorwort

Dieses Handbuch richtet sich an Entwickler, die mit der *zen Platform* der zeos informatics GmbH serverbasierte Java-Anwendungen erstellen wollen. Nach der ausführlichen Vorstellung von Konzept und Arbeitsweise der *zen Platform* (Kapitel 1 – 4) wird die Benutzung der Entwicklungsumgebung (Kapitel 5) erläutert. In einer Referenz werden die Schnittstellen zur Anwendungsentwicklung (Kapitel 6), praxisnahe Entwurfsmuster (Kapitel 7) und die Administration der Laufzeitumgebung (Kapitel 8) vorgestellt. Zum Abschluß folgt die Konfiguration bzw. das Deployment der *zen Platform* (Kapitel 9 – 10).

In diesem Handbuch wird das Instrumentarium für alle Aufgaben vermittelt, die im Rahmen des Entwicklungsprozesses mit der *zen Platform* anfallen, wie z.B.:

- Entwurf
- Entwicklung und Test
- Deployment und Konfiguration
- Überwachung und Pflege

1 Die zen Platform

Die *zen Platform* ist eine J2EE-Entwicklungs- und Laufzeitumgebung. Sie besteht aus der Entwicklungsumgebung *zen Developer* und dem Laufzeitsystem *zen Engine*. Die *zen Platform* vereinfacht und beschleunigt die Entwicklung serverbasierter, auch geschäftskritischer Java-Anwendungen und erleichtert die Einarbeitung in Projekte gleichermaßen wie deren Pflege und Wartung. So werden zeitliche Aufwände sowie Kosten von Entwicklung und Betrieb einer Anwendung erheblich reduziert.

Die *zen Platform* vereinfacht die technische Komplexität der J2EE-Umgebung und bietet eine weitreichende Unterstützung für die Implementierung der Geschäftslogik. Durch die lösungs-orientierte Arbeitsmethodik kann sich der Entwickler fast ausschließlich auf die Umsetzung der Geschäftslogik zu konzentrieren. Mit diesem eigentlich selbstverständlichen Ansatz steht die Lösung von Geschäftsproblemen wieder im Mittelpunkt der Entwicklung.

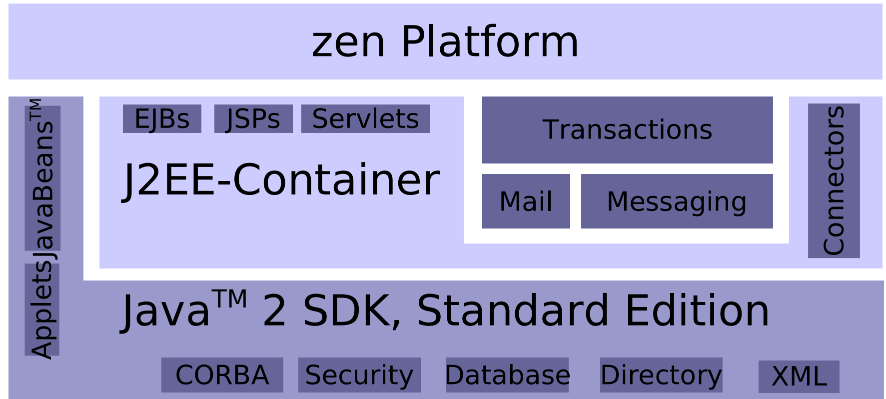

Abbildung 1: zen Platform im J2EE-Kontext

Die *zen Platform* unterstützt die Entwicklung von Anwendungen mit Web-Frontends ebenso wie Web-Services und Anwendungen mit beliebigen anderen Schnittstellen.

1.1 Die Entwicklungsumgebung

Der Entwicklungsprozeß der *zen Platform* wird mit modernen visuellen Techniken im *zen Developer* durchgeführt. Der *zen Developer* ist als Plugin in die weit verbreitete und freie Java-IDE Eclipse integriert.

Eine *zen*-Anwendung kann mit dem *zen Developer* ohne technisches Expertenwissen graphisch modelliert werden. Unterstützt wird die Datenmodellierung, die graphische Modellierung der Ablaufsteuerung (Workflow), das Verknüpfen von Geschäftslogik mit Daten und Workflow, sowie die Fehlerrecherche (Debugging). Die so entstandenen Anwendungsmodelle werden dann in einer beliebigen Datenbank abgelegt.

1.2 Das Laufzeitsystem

Das Laufzeitsystem der *zen Platform* ist die *zen Engine*. Sie liest die mit dem *zen Developer* erstellten Anwendungsmodelle aus der Datenbank (dem Repository) und kann dann Anfragen von Clients entgegennehmen. Ein Client kann beispielsweise ein Web-Browser oder eine beliebige Anwendung sein, die mit einer *zen*-Anwendung über eine Web-Service-Schnittstelle kommuniziert. Die Anfrage wird von der *zen Engine* abhängig vom jeweiligen Workflow- und Datenzustand und der damit verknüpften Geschäftslogik beantwortet.

Grundvoraussetzung für den Einsatz der *zen Engine* ist ein beliebiger Servlet-Container. Dieser ist bei vielen (nicht unternehmenskritischen) Anwendungen für den produktiven Einsatz ausreichend. Die *zen Engine* abstrahiert von den technischen Schnittstellen der EJB-Spezifikation, daher kann sie ebenso über einen Cluster von Applikationsservern skalieren; hierzu muß nur die Konfiguration der *zen Engine* angepaßt werden, die Modelle und die Anwendungslogik werden durch die Skalierung nicht beeinflußt.

Der Entwickler kann bei der Implementierung der Geschäftslogik auf einen umfangreichen Katalog von Diensten (Services) zurückgreifen, die ihm über einfach zugängliche Schnittstellen von der *zen Engine* zur Verfügung gestellt werden. Die Möglichkeiten gehen dabei deutlich über die üblicherweise verfügbaren Services der EJB-Spezifikation hinaus.

Alle Anwendungen, die von einer *zen Engine* ausgeführt werden, können über eine zentrale Management-Konsole konfiguriert, überwacht und sogar während des Betriebs erweitert und korrigiert werden.

2 Überblick über die *zen Engine*

Die *zen Engine* ist als Laufzeitsystem der Kern der *zen Platform.* Sie liest die Anwendungsmodelle aus dem Repository und interpretiert diese zur Laufzeit. Als Middleware bietet sie für die verfügbaren Services eine einheitliche abstrakte Schnittstelle – unabhängig von der zugrundeliegenden technischen Umgebung. Außerdem verteilt sie die Anfragen bei einem Cluster-Deployment und überwacht das Laufzeitsystem (Monitoring).

In einer *zen*-Anwendung werden wesentliche Bestandteile der Anwendung nicht starr implementiert, sondern stattdessen graphisch mit dem *zen Developer* modelliert. Ablaufsteuerung (Workflow), Datendefinition, Verknüpfung mit Geschäftslogik und weitere fundamentale Anwendungseigenschaften sind somit nicht fest verdrahtet, sondern liegen als Modell vor, das in einer Datenbank (Repository) abgelegt wird. Durch das graphische Modell erschließt sich nicht nur die Funktionsweise der Anwendung auf einfache und intuitive Weise, auch Erweiterungen und Wartungsarbeiten werden so deutlich erleichtert.

Die *zen Engine* liest zur Laufzeit das Modell einer *zen*-Anwendung aus dem Repository und erstellt ein dynamisches Ablauf- und Datenmodell der Anwendung. Das Einlesen erfolgt erst bei Bedarf; einmal gelesene Informationen werden von der *zen Engine* in einem Cache gespeichert. Anhand des Laufzeitmodells steuert die *zen Engine* dann die gesamte Anwendung. Die *zen Engine* nimmt Anfragen von Clients entgegen und entscheidet anhand des Laufzeitmodells, ob die Anfrage korrekt ist, wie sie bearbeitet und beantwortet wird.

Wenn sowohl *zen Engine* als auch *zen Developer* auf die gleiche Repository-Instanz zugreifen, kann das Anwendungsmodell sogar während der Laufzeit verändert werden.

2.1 Architektur der zen Engine

Die *zen Engine* ist logisch unterteilt in ein Frontend und ein Backend. Das Frontend übernimmt die Abstraktion von technischen Aufgaben, das Backend ist für die Workflowsteuerung und Datenverwaltung verantwortlich und arbeitet unabhängig von spezifischen technischen Gegebenheiten. Frontend und Backend kommunizieren über eine offene Schnittstelle.

- **Frontend**: Ein Frontend ist für eine bestimmte technische Umgebung spezifiziert und bildet die Schnittstelle zur Außenwelt. Beispielsweise verhält sich das Web-Frontend für browserbasierte Anwendungen wie ein normales Servlet, das Anfragen von einem Web-Browser entgegennimmt und beantwortet. Das Web-Service-Frontend kommuniziert hingegen mit einem Client über SOAP. Das EJB-Frontend kann benutzt werden, um die *zen Engine* von anderen serverseitigen Komponenten aus aufzurufen.

  Das jeweilige Frontend wandelt die Anfragen der verschiedenen Clients in das Format der Frontend-Backend-Schnittstelle um und ruft anschließend das Backend auf. Ebenso wird vom Frontend die Antwort des Backends wieder zurück in das jeweils erwartete Ausgabeformat umgewandelt.

- **Backend**: Das Backend ist für die eigentliche Workflowsteuerung- und Datenverwaltung verantwortlich. Nur das Backend hat Zugriff auf das Repository und kennt so das Modell der jeweiligen Anwendung. Es nimmt die Anfrage des Frontends entgegen, überprüft diese mit Hilfe des Modells auf Korrektheit, verknüpft die Anfrage-Daten mit der Session und arbeitet, abhängig vom jeweiligen Session-Zustand, den Workflow und die Geschäftslogik weiter ab. Anschließend schickt es die entsprechende Antwort zurück an das Frontend.

Durch diese Trennung kann eine Anwendung für unterschiedliche technische Umgebungen erstellt werden, ohne daß die Anwendung selbst auf die technischen Unterschiede eingehen muß. Insbesondere können so auch verschiedene Ein- und Ausgabeformate für eine Anwendung transparent und parallel genutzt werden.

Eine zen-Anwendung, die beispielsweise Börsenorders entgegennimmt und eine Liste der laufenden Orders zurückgibt, kann z.B. parallel und ohne Modifikation mittels verschiedener Frontends:
- ein Web-Formular implementieren, das Orderdaten abfragt und eine Orderliste präsentiert
- als Web-Service in Makleranwendungen eingebettet werden
- in einer Call-Center-Anwendung benutzt werden, um Orders entgegenzunehmen und die Orderliste als PDF-Dokument für den postalischen Versand zu erzeugen

---

Zusätzlich zu den genannten Möglichkeiten lassen sich beliebige weitere Frontends gegen die Frontend-Backend-Schnittstelle entwickeln, die praktisch unbegrenzte Möglichkeiten bezüglich Schnittstellen, Protokollen und Datenformaten ermöglichen, ohne die *zen*-Anwendung modifizieren zu müssen.

2.2 Funktionsweise

Die *zen Engine* arbeitet grundsätzlich nach dem Request-Response-Modell. Eine *zen*-Anwendung kann dabei, sofern erforderlich, auf umfassendes Sessionmanagement zurückgreifen. Im folgenden wird zur Verdeutlichung der Funktionsweise der *zen Engine* ein Request-Response-Zyklus am Beispiel eines Web-Browser-Requests erläutert.

Das Frontend der *zen Engine* für browserbasierte Anwendungen ist als Servlet realisiert. Das Servlet nimmt Anfragen von Web-Browsern entgegen und transformiert die vom Browser mitgelieferten Daten (z.B. Inhalte eines Web-Formulars) für die Frontend-Backend-Schnittstelle in eine hierarchische XML-kompatible Datenstruktur (FOM API, Flexible Object Model). Zusätzlich synchronisiert das Web-Browser-Frontend das Sessionmanagement der Servlet API mit dem der *zen Engine*. Dann können die Daten an das Backend geschickt werden.

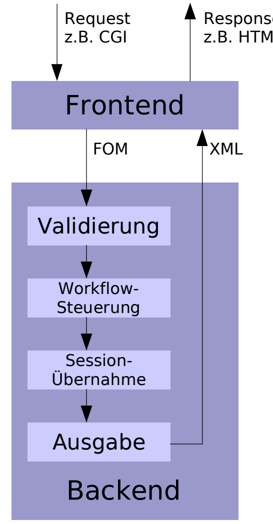

Abbildung 2: Schematischer Aufbau zen Engine

Im Backend werden nun die Daten mit dem Anwendungsmodell verglichen. Mittels der 5-Level-Validierung wird überprüft, ob der aktuelle Workflowzustand gültig ist, ob die mitgelieferten Daten auf den Workflowzustand passen, ob die Dateninhalte mit den definierten Datentypen übereinstimmen, ob Wertedomänen gültig sind und Geschäftsregeln eingehalten wurden. Anhand des Anwendungsmodells wird der Übergang zum nächsten Workflowzustand bestimmt. Zusätzlich wird die mit dem Workflow und den Daten verknüpfte Geschäftslogik ausgeführt, wodurch die Daten eventuell mit Arbeitsergebnissen angereichert werden. Im Erfolgsfall werden die Daten anschließend von der *zen Engine* in die Session übernommen. Zum Abschluß wird die Ausgabe anhand des nächsten Workflowzustands im XML-Format erzeugt und an das Frontend zurückgeschickt.

Das Frontend nimmt die XML-Antwort des Backends entgegen und ist dafür verantwortlich, diese in das nötige Ausgabeformat zu transformieren. Das Web-Browser-Frontend nutzt hierzu einen XSL-Prozessor, mit dem die Ausgabe in der Regel nach HTML konvertiert wird. Da das Zielformat durch das XSL-Stylesheet festgelegt wird, können je nach Stylesheet auch andere Formate erzeugt werden, wie z.B. PDF oder Text. Die so erzeugte Ausgabe wird schließlich als Response an den Web-Browser gesendet.

Außerhalb des Web-Bereichs funktioniert der Ablauf im wesentlichen analog. Wird z.B. das EJB-Frontend benutzt, kann man sich den Request-Response-Zyklus als Methodenaufruf mit Rückgabe vorstellen.

2.3 Services

Die *zen Platform* stellt eine Serviceschicht zur Verfügung, die zur Implementierung der Geschäftslogik genutzt werden kann. Die Services werden über die SCF-Service-API (Scalable Component Framework) angebunden. Diese umfaßt neben verschiedenen Services der EJB-Spezifikation eine Reihe von zusätzlichen Services der *zen Platform.* Alle Services können transparent genutzt werden, egal, ob die *zen*-Anwendung in einer Servlet Engine oder einem Applikationsserver-Cluster läuft.

- **Session Management**: Über den Session Management Service kann direkt auf die aktuelle Session zugegriffen werden, um beliebige eigene Daten abzulegen. Das Session Management kann sich automatisch mit anderen Rechnern synchronisieren, wenn die *zen Platform* in einem Cluster läuft.
- **Logging**: Man kann eigene Log-Ziele definieren (Datei, Datenbank, Mail, etc.) und auf verschiedenen Warnstufen loggen. Die zugrundeliegenden Logsysteme lassen sich zudem beliebig erweitern.
- **Connection**: Der Connection Service stellt Verbindungen auf beliebige Datenbanken über die JDBC-API zur Verfügung.
- **JDO**: Der JDO Service stellt den Zugriff auf die Persistenzschicht über die *Java Data Objects* Spezifikation zur Verfügung. Die *zen Platform* nutzt den JDO-Service, um auf das Repository der einzelnen *zen*-Anwendungen zuzugreifen.
- **Transaction**: Der Transaction Service (JTA) sorgt für Datenkonsistenz auf transaktionalen Ressourcen.
- **Mail**: Der Mail Service kapselt den Zugriff auf den Mail-Server.
- **Resource Repository**: Das Resource Repository dient zum Zugriff (location-transparent) auf beliebige Text- oder Binär-Ressourcen. Ein Resource Repository kann unter anderem das Dateisystem, die Datenbank oder ein Http-Server sein.
- **Messaging**: Der Messaging Service bietet nicht nur Unterstützung für Fire-and-Forget-Messages, sondern auch für überwachte Messages.
- **Component Selector**: Über den Component Selector Service kann man (location-transparent) auf SCF-Komponenten zugreifen.

Nähere Informationen zur Verwendung der Services sind in Kapitel 6 (zen Engine Referenz) zu finden.

---

3 Aufbau einer *zen*-Anwendung

Eine *zen*-Anwendung wird mit dem *zen Developer* weitgehend graphisch modelliert. Das Modell umfaßt im wesentlichen ein Workflowmodell und ein Datenmodell. Im Workflowmodell wird die Ablaufsteuerung der Anwendung festgelegt. Im Datenmodell wird definiert, welche Daten für Ein- und Ausgaben zur Verfügung stehen bzw. mit Geschäftslogik verknüpft werden können.

3.1 Workflowmodell

Die *zen Engine* arbeitet nach dem Request-Response-Prinzip. Der Workflow einer *zen*-Anwendung besteht aus der Menge aller möglichen Requests/Responses, die bestimmte Zustände innerhalb der Anwendung miteinander verbinden.

Eine Web-Anwendung nimmt Börsenorders in einem Formular entgegen. Nach Eingabe der Orderdaten bekommt der Nutzer eine Seite mit allen laufenden Orders präsentiert. Liegt das Formular nicht als statische HTML-Seite vor, sondern wird bereits von der Anwendung dynamisch erstellt, kann man drei Zustände und zwei Requests/Responses identifizieren. Wir haben einen Startzustand, von dem aus die Anwendung gestartet wird. Mit einem initialen Request wird der Startzustand verlassen und ein Zustand eingenommen, der als Response das Formular für die Kundendaten erzeugt. Das Formular wird mit dem zweiten Request abgeschickt und die Formular-Daten können entgegengenommen werden. Die Geschäftslogik fügt die Order zu den zur Zeit laufenden Orders hinzu und der Endzustand wird eingenommen, indem die Anwendung die laufenden Orders anzeigt.

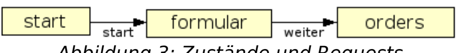

Abbildung 3: Zustände und Requests

Der Workflow einer *zen*-Anwendung wird als Zustandsübergangsdiagramm modelliert. In Web-Anwendungen können Zustände in der Regel als Seiten verstanden werden, in Anwendungen ohne Visualisierung als Eingabe- und Ausgabeschnittstellen. Die Zustände bilden die Knoten (State Nodes) im Zustandsübergangsdiagramm. Sie werden mit Kanten (Transitions) miteinander verbunden; diese kann man sich als Request vorstellen. Um die verschiedenen möglichen Transitions zu unterscheiden, die von einem State Node ausgehen können, werden sie mit einer Action bezeichnet (z.B. in Abbildung 3: die Transition vom State Node *formular* zum State Node *orders* wird mit der Action *weiter* bezeichnet). Eine Action identifiziert also zusammen mit dem Quell-State-Node einen Request. Eine Action kann man sich im Web-Umfeld z.B. als Klick auf einen bestimmten Button eines Formulars vorstellen: Verschiedene Buttons auf einem Formular können zu verschiedenen Folgezuständen führen. Schließlich kann sich eine Transition unter Umständen auffächern: Sie kann von einem Zustand aus in verschiedene Folgezustände verzweigen. Abhängig von weiteren Daten wird während der Abarbeitung des Workflows ein Zustand aus der Menge der Folgezustände ausgewählt. Hierzu wird ein zusätzlicher Knotentyp, ein Decision Node, verwendet.

Zusammengefaßt wird im *zen Developer* ein Workflow also mit Hilfe folgender Elemente modelliert:

- **State Nodes**: Ein State Node stellt einen Zustand innerhalb der Anwendung dar
- **Decision Nodes**: Ein Decision Node wird über eine Transition von einem State Node aus betreten und verzweigt abhängig von weiteren Daten zu beliebig vielen Folge-State Nodes
- **Transitions**: State Nodes und Decision Nodes werden über Transitions miteinander verbunden
- **Actions**: Eine Action identifiziert eine abgehende Transition eines State Nodes

Der Workflow einer *zen*-Anwendung kann in mehrere (Teil-)Prozesse unterteilt werden. Jedem Prozess wird dann ein Ausschnitt aus dem Workflow zugeordnet. Damit lassen sich komplexe Workflows gliedern.

Wir erweitern nun das vorige Beispiel der Web-Anwendung um zusätzliche Workflow-Elemente. Vom Eingabeformular soll nicht mehr direkt auf die Orderliste verzweigt werden. Stattdessen wird zusätzlich überprüft, ob das Ausführungslimit der Order eingehalten werden kann. Ist das nicht der Fall, soll der Nutzer einen Hinweis erhalten, daß die Order ohne Limit ausgeführt wird. Wird dies akzeptiert, kann die Order ausgeführt werden. Hierzu führen wir einen Decision Node ein, der die Entscheidung über das Orderlimit trifft. Ist das Limit gültig, wird auf die Orderliste verzweigt. Sonst wird ein neuer State Node mit dem Warnhinweis angezeigt. Zusätzlich werden neue Transitions eingeführt: Vom Warnhinweis auf die Orderliste bzw. zurück auf das Eingabeformular und von der Orderliste zurück auf das Eingabeformular. In Abbildung 4 sehen wir den kompletten Workflow.

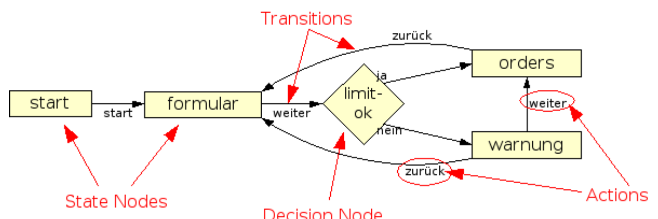

Abbildung 4: Workflow der Beispiel-Anwendung

3.2 Datenmodell

Das Datenmodell einer *zen*-Anwendung wird ebenfalls graphisch modelliert und im Repository abgelegt. Es beinhaltet alle Eingabe- und Ausgabedaten einer Anwendung, sowie alle Daten, die mit Geschäftslogik verknüpft werden können. Dieses Datenmodell kann das vollständige Datenmodell für die gesamte Anwendung darstellen.

Zur Laufzeit wird das Datenmodell über die dynamische hierarchische FOM API (Flexible Object Model) abgebildet. In dem FOM-Modell werden alle Eingabedaten abgelegt und von diesem werden alle Ausgabedaten ausgelesen.

Bei der Entwicklung herkömmlicher Java-Anwendungen verwendet man üblicherweise komplexe Business-Objekte. Diese bilden die entsprechende Daten-Domain ab, werden komplett in Java implementiert und müssen vor der Verwendung kompiliert werden.

Das Datenmodell einer *zen*-Anwendung ist ein streng hierarchisches, an XML angelehntes Datenmodell. Ausgehend von einem festgelegten Wurzelelement wird das Modell auf Basis der elementaren Datenstrukturen Atom, Komposition (Composition) und Liste (List) definiert.

- **Atom**: Ein Atom ist ein Blatt des hierarchischen Datenmodells. Als Atom werden tatsächliche Werte wie String, Integer oder eigene Datenformate abgebildet. Jedem Atom wird hierzu ein Datentyp zugewiesen.
- **Composition**: Eine Composition ist eine Aggregation beliebiger Elemente.
- **List**: Eine List wird auf einer Composition oder einem Atom definiert. Zur Laufzeit kann die List dann eine Menge der definierten Elemente gleichen Typs enthalten.

So entsteht ein sehr einfaches, verständliches und doch leistungsfähiges Modell, das sogar zur Laufzeit verändert werden kann.

Konstruieren wir ein Datenmodell für die Beispiel-Anwendung. Das Datenmodell umfaßt mindestens alle Daten, die als Eingabe- bzw. Ausgabedaten vorkommen können. In unserem Fall sind das die Orderdaten, die in einem Eingabeformular erfaßt werden, und die Daten für die Orderliste, die danach angezeigt wird. Diese Daten besitzen verschiedene Datentypen und werden als Atoms modelliert. Wir nutzen Compositions und Lists zur Strukturierung. Wir fassen z.B. die Orderdaten unter einer Composition order *zusammen. Die Daten der laufenden Orders fassen wir unter einer Composition* lfd-order *zusammen und definieren eine List* lfd-orders*, die aus* lfd-order*-Compositions bestehen kann. Im Modell hat eine List immer nur ein Kind, da es sich hier sozusagen um die Typdefinition handelt. In der Laufzeitabbildung kann die List dann aus beliebig vielen* lfd-order*-Compositions bestehen. Schließlich definieren wir noch die Composition* depot*, die nur eine Depotnummer enthält (Berechtigungssysteme oder eine Depotverwaltung werden hier nicht beachtet).*

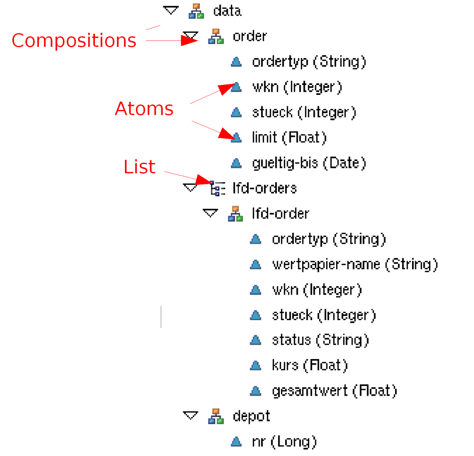

Abbildung 5: Beispiel-Datenmodell

3.3 Verknüpfung von Workflow und Daten

Sind Workflow- und das Datenmodell definiert, werden beide Modelle miteinander verknüpft. Für jeden State Node des Workflows wird festgelegt, welche Elemente aus dem Datenmodell als Eingabe- und/oder Ausgabedaten genutzt werden.

In der Beispiel-Anwendung ordnen wir dem State Node formular *die gesamte Composition* order *als Ein- und Ausgabe zu. Alle Felder, die als Eingabe zugeordnet sind, können von diesem State Node an die Anwendung geschickt werden, liegen also als Formular-Felder vor. Als Ausgabe müssen sie insbesondere deshalb zugeordnet werden, da das Formular anhand des Datenmodells angezeigt und die Inhalte anhand vorhandener Daten vorbelegt werden sollen (wenn der Nutzer z.B. von dem Warnhinweis zurück zum Formular wechselt, sollen die zuvor eingegebenen Daten im Formular wieder angezeigt werden).*

Dem State Node orders *ordnen wir die List* lfd-orders *als Ausgabe zu. Zusätzlich ordnen wir dem State Node* orders *noch das Atom* nr *aus der Composition* depot *zu, damit die Depotnummer angezeigt werden kann. Solange in diesem State Node nur Text angezeigt wird und der Nutzer keine Selektion oder weiteren Eingaben vornehmen kann, ordnen wir keine Elemente als Eingabe zu.*

Zum Schluß werfen wir noch einen Blick auf den State Node start*. Dieser wird in unserem Beispiel nie zur Anzeige genutzt. Er könnte dennoch Eingabedaten enthalten, z.B. wenn unsere Anwendung immer als Link mit der Depotnummer aufgerufen werden würde. In diesem Fall würden wir dem State Node* start *das Atom* nr *aus der Composition* depot *als Eingabe zuordnen.*

Ein so verknüpftes Workflow- und Datenmodell stellt bereits einen ersten, funktionstüchtigen Anwendungsprototypen dar. Mit dem mitgelieferten generischen XSL-Stylesheet kann der Workflow in einer Basisdarstellung mit automatisch generierten Eingabefeldern durchlaufen werden und als Diskussionsgrundlage für die weitergehende Entwicklung dienen.

3.4 Geschäftslogik

Die Implementierung der geschäfts- und anwendungsspezifischen Logik, die über die Workflow- und Datenmodellierung hinausgeht, unterstützt die *zen Platform* mit einer Vielzahl von effektiven Diensten und automatischen Mechanismen. Dies vereinfacht die Verwendung auch komplexer J2EE-Technologien wesentlich.

Die Geschäftslogik wird in Java implementiert. Im *zen Developer* wird ein Einsprungpunkt in den Java-Code als sogenannte Operation modelliert. Operations können dann mit dem Workflow- und/oder Datenmodell verknüpft werden. Dadurch bestimmt sich einerseits der Ausführungszeitpunkt jeder Operation, andererseits wird so definiert, welche Daten den Operations jeweils zur Verfügung stehen bzw. von diesen verändert werden können.

Die *zen Platform* unterscheidet zwei Kategorien von Operations:

- **Workflow Operations**: Workflowgesteuerte Operations werden vom Entwickler an bestimmte Workflowelemente, z.B. an State Nodes, Transitions oder Actions gekoppelt. Dadurch wird der Ausführungszeitpunkt bestimmt
- **Business Rules**: Business Rules werden nur mit dem Datenmodell verknüpft und immer dann automatisch ausgeführt, wenn die verknüpften Datenelemente als (geänderte) Eingabedaten vorliegen.

3.4.1 Workflow Operations

Innerhalb eines Workflows können Operations an verschiedene Workflow-Elemente gekoppelt werden. An ein Workflow-Element können in der Regel beliebig viele Operations in einer vom Entwickler vorgegebenen Reihenfolge gehängt werden. Die verschiedenen Arten von Workflow Operations sind:

- **Pre State Operations**: sind einem State Node zugeordnet. Sie werden aufgerufen, bevor der entsprechende State Node betreten wird.
- **Post State Operations**: sind ebenso einem State Node zugeordnet. Sie werden aufgerufen, wenn der entsprechende State Node verlassen wird.
- **Action Operations**: sind einer Action zugeordnet. Sie werden aufgerufen, wenn eine Action ausgelöst wurde. Danach wird die zugehörige Transition betreten.
- **Transition Operations**: sind einer Transition vom einem State Node zu einem beliebigen anderen Node zugeordnet. Sie werden nach den Action Operations aufgerufen, wenn die Transition betreten wird.
- **Decision Operations**: sind einem Decision Node zugeordnet. Sie entscheiden dynamisch, wohin der Workflow nach dem Decision Node hin verzweigt. Es kann pro Decision Node nur eine Decision Operation geben, die das Entscheidungskriterium bestimmt.
- **Post Decision Operations**: sind der Transition von einem Decision Node zu einem State Node zugeordnet. Sie werden aufgerufen, wenn zuvor eine Decision Operation entschieden hat, mit der entsprechenden Transition fortzufahren.

In unserem Depot-Beispiel benötigen wir eine Reihe von Workflow Operations, z.B. für die tatsächliche Durchführung der Order, die Limit-Entscheidung für den Decision Node, eine Datenbankspeicherung der bisherigen Orders und einige mehr. Bis auf die Decision Operation gibt es für die meisten Workflow Operations mehrere Möglichkeiten der Verknüpfung. Die Durchführung der Order könnte z.B. in die „ja“-Transition des Decision Node limit-ok *und zusätzlich in die „weiter“-Transition des State Nodes* warnung *gehängt werden, oder stattdessen als Pre State Operation an den State Node* orders *gehängt werden.*

3.4.2 Business Rules

Business Rules sind nur mit dem Datenmodell verknüpft. Die Entscheidung über die Ausführung einer Business Rule wird anhand der Verknüpfung mit dem Datenmodell und dem jeweils aktuellen Zustand der Session-Daten getroffen. Der Aufruf erfolgt nur, wenn sich Eingabedaten, die der Business Rule zugeordnet wurden, gegenüber den Session-Daten geändert haben. Die Reihenfolge der Ausführung wird automatisch anhand der Abhängigkeiten zwischen den Business Rules bestimmt, die durch die Analyse der Eingabe- und Ausgabeparameter der Business Rules ermittelt werden. Jeder Business Rule kann zusätzlich eine Priorität zugeordnet werden. Damit wird bei unabhängigen Business Rules die Ausführungsreihenfolge festgelegt.

- **Computation Rules**: sind Business Rules, die nach fehlerfreier Basis-Validierung der Eingabedaten ausgeführt werden. Sie dienen dazu, neue Daten zu berechnen oder zu verändern. Die Ausgabeparameter von Computation Rules werden nach Ausführung in die aktuelle Datenstruktur übernommen und können weitere Business Rules auslösen, wenn diese die entsprechenden Daten wieder als Eingabeparameter definiert haben
- **Validation Rules**: sind Business Rules, die in erster Linie für die Validierung von Geschäftsregeln verantwortlich sind. Sie haben keine Ausgabeparameter, sondern signalisieren der *zen Engine* über Exceptions, wenn Geschäftsregeln verletzt wurden.

In unserem Beispiel könnten wir z.B. eine Operation als Computation Rule definieren, die anhand der Wertpapierkennnummer den Wertpapiernamen ermittelt und in das Datenmodell einfügt. Wann immer die Wertpapierkennnummer vom Nutzer eingegeben oder verändert wird und an die zen Engine geschickt wird, wird die Computation Rule erneut aufgerufen.

Da Computation und Validation Rules nur mit dem Datenmodell verknüpft sind, verschaffen sie dem Entwickler eine gewisse Unabhängigkeit gegenüber dem Workflowmodell. Sie erzielen insbesondere in Anwendungsfällen einen Mehrwert, in denen Workflows häufig geändert werden oder in denen gleiche Daten in verschiedenen Zuständen als Eingabedaten auftreten können.

3.5 Ressourcen und Attribute

Das Workflow- und Datenmodell kann mit sprachspezifischen Ressourcen und manuell definierten Attributen versehen werden.

Ressourcen

Sprachspezifische Ressourcen sind z.B. Texte oder Pfade für sprachspezifische Bilder. Ressourcen werden während der Modellierung mit verschiedenen Modellelementen verknüpft. Zur Laufzeit fügt die *zen Engine* solche Ressourcen der jeweiligen XML-Antwort des Backends an das Frontend hinzu. Das XSL-Stylesheet kann diese Ressourcen bei der Transformation der XML-Antwort in das endgültige Ausgabeformat zur inhaltlichen Gestaltung nutzen.

Folgende Modellelemente können mit Ressourcen verknüpft werden:

- **State Nodes:** z.B. Texte und Bilder zur Seitengestaltung von Anwendungen.
- **Actions:** z.B. Aufschriften oder Bilder für HTML-Buttons
- **Datenmodell-Elemente:** z.B. Labels für Eingabefelder
- **Operations:** Fehlermeldungen für den Endnutzer aus Operationen (Operation Errors)
- **Datatypes, Domains, Elemente:** Fehlermeldungen für den Endnutzer bei verschiedenen Arten von Eingabefehlern (Data Errors)

Attribute

Attribute sind manuell definierte, sprachunabhängige Zusatzinformationen, die mit verschiedenen Modellelementen verknüpft werden. Zur Laufzeit werden diese Attribute – wie Ressourcen – den ausgegebenen Modellelementen hinzugefügt. Das XSL-Stylesheet kann Attribute z.B. als Hinweis zur optischen Gestaltung oder zur Formatierung der Ausgabe nutzen. Anwendungen ohne Benutzeroberfläche können Attribute z.B. als Zusatzhinweise zur Interpretation von Schnittstellen verwenden.

Folgende Modellelemente können mit Attributen verknüpft werden:

- **State Nodes:** Zusatzinformationen zur Seitengestaltung
- **Actions:** Zusatzinformationen zur Gestaltung von Actions
- **Datenmodell-Elemente:** Zusatzinformationen zur Gestaltung von Eingabeelementen

4 Ablauf einer zen-Anwendung

Der Ablauf einer *zen*-Anwendung kann auf zwei Ebenen betrachtet werden. Auf Ebene eines gesamten Workflows ergibt sich der Ablauf aus dem Workflow-Modell. Auf Ebene eines einzelnen Requests ist der Bearbeitungsablauf einer Anfrage prinzipiell durch die *zen Engine* vorgegeben. Über die Einstellung verschiedener Eigenschaften im Anwendungsmodell kann dieser jedoch an die jeweiligen Erfordernisse angepaßt werden.

4.1 Grundlagen

Zunächst werden einige grundlegende Begriffe erläutert, die im folgenden häufig verwendet werden.

Request-Daten

Ein Request an eine *zen*-Anwendung enthält Daten, die von der Anwendung bearbeitet werden sollen. In welcher Form diese Daten vorliegen, ist abhängig vom Frontend. Diese Daten lassen sich jedoch – unabhängig vom Frontend – immer in folgende drei Kategorien unterteilen:

- **Kontrollflußinformationen**: Werden der *zen Engine* mitgeteilt, um den Workflow zu steuern. Sie identifizieren den Request, bzw. gleichen diesen mit dem Sessionzustand ab. Die Kontrollflußinformationen müssen vom Frontend in einem fest definierten Schnittstellenformat zur Verfügung gestellt werden.
    - **State Node**: Die *zen Engine* erwartet in einem Request die Angabe des gewünschten bzw. aktuellen State Nodes. Für eine neue Session dient dieser zur Identifikation des Einsprungpunktes in den Workflow. Wird der State Node beim Start eines Workflows weggelassen, wird automatisch der zentrale Einsprungpunkt des Workflows ausgewählt. Für eine laufende Session wird der im Request angegebene State Node aus Konsistenzgründen mit dem zuletzt betretenen State Node verglichen, der in der Session gespeichert ist.
    - **Action**: Die *zen Engine* erwartet im Request die Angabe der Action, die den Request ausgelöst hat. Die Action bestimmt, welche Transition von einem State Node aus betreten wird. Besitzt ein State Node nur eine einzige ausgehende Transition, kann die Angabe der Action entfallen.
    - **Spracheinstellungen**: Optionale Kontrollflußinformation, die angibt, in welcher Spracheinstellung der Request bearbeitet wird. Hat z.B. Einfluß auf die Validierung von länder- oder sprachspezifischen Daten bzw. auf deren Ausgabe.
- **Modellierte Anwendungsdaten**: Geschäftsspezifische Daten, mit denen die *zen*-Anwendung in erster Linie arbeitet. Die Schnittstelle ergibt sich direkt aus dem Datenmodell einer *zen*-Anwendung. Diese Daten werden z.B. validiert und können in Operations verwendet werden.
- **Nicht-modellierte Anwendungsdaten**: Technische anwendungsspezifische Daten für Sonderfälle. Sie sind nicht im Datenmodell definiert, die Schnittstelle ist offen. Sie werden nicht validiert und können nur eingeschränkt in Operations verwendet werden.

Fehler- und Ausnahmefälle

Wird eine Anfrage an eine *zen*-Anwendung bearbeitet, können verschiedene Arten von Fehler- und Ausnahmefällen eintreten, die Einfluß auf die weitere Bearbeitung haben können. Wir unterscheiden im folgenden zwischen den Kategorien:

- **Benutzerfehler**: Entstehen z.B. aufgrund fehlerhafter Benutzereingaben in ein Web-Formular.
- **Konsistenzfehler**: Können aufgrund des Benutzerverhaltens entstehen, z.B. bei gleichzeitiger Nutzung einer Anwendung in zwei Browser-Fenstern, wenn mit Cookies gearbeitet wird, bei Nutzung der Browser-Funktionen Vorwärts/Zurück/Reload, o.ä.
- **Timeout**: Entsteht, wenn die Gültigkeitsdauer einer Session abgelaufen ist.
- **Anwendungsfehler**: Entstehen z.B. bei falscher Nutzung der Schnittstelle durch Clients, fehlerhaft modellierte Workflows oder bei Fehlern im Anwendungscode.
- **Kritische Fehler**: Entstehen durch Systemfehler (z.B. Repository nicht verfügbar) oder bei Anfragen, die keiner Anwendung zugeordnet werden können.

Action Types

Jeder Action wird im Workflowmodell ein Action Type zugeordnet. Diese Eigenschaft bestimmt wesentlich, wie eine Anfrage bearbeitet wird:

- **Default**: Eine Transition mit einer *default*-Action ist der Normalfall. Andere Action Types werden nur in Ausnahmefällen gebraucht.
- **Cancel**: Bei einer Transition mit einer *cancel*-Action werden die Benutzereingaben weder validiert noch in die Session übernommen.
- **Clear**: Bei einer Transition mit einer *clear*-Action werden die Benutzereingaben nicht validiert und anschließend aus der Session gelöscht.
- **Nonvalidating**: Bei einer Transition mit einer *nonvalidating*-Action werden Benutzereingaben nicht validiert, jedoch in die Session übernommen.

- **Erroraware**: Eine Transition mit einer *erroraware*-Action bricht bei Anwendungsfehlern die Bearbeitung nicht ab. Die Fehler werden stattdessen aggregiert. Bei Benutzerfehlern werden alle Bearbeitungsschritte weiter ausgeführt.
- **Terminal**: Eine Transition mit einer *terminal*-Action kann parallel zum aktuellen Workflow laufen, Start- und Endzustände einer solchen Transition werden nicht auf Workflowkonsistenz geprüft.

State Gates

Jedem State Node wird die Gate-Eigenschaft zugeordnet, die die Konsistenzprüfung bei der Workflow-Validierung steuert und Einsprung- bzw. Aussprungpunkte des Workflows festlegt:

- **Default**: Normaler State Node im Workflow.
- **Entry**: Einsprungpunkt in den Workflow.
- **Defaultentry**: Zentraler standardmäßiger Einsprungpunkt in den Workflow. Wird die Anwendung ohne Angabe eines State Nodes aufgerufen, wird der Workflow mit dem so gekennzeichneten State Node begonnen. Nur ein einziger State Node im Workflow darf so gekennzeichnet sein.
- **Exit**: Aussprungpunkt aus dem Workflow. Befindet sich ein Client auf einem so gekennzeichneten State Node, wird dessen Workflow als abgeschlossen betrachtet, was zu einer schnelleren Session-Löschung führen kann und somit Wartungsarbeiten vereinfacht.

4.2 Überblick

Die Bearbeitung eines Requests wird zunächst nur kurz skizziert und dann in den nachfolgenden Abschnitten im Detail erläutert.

Anhand der im Request vorliegenden Kontrollflußinformation stellt die *zen Engine* fest, von welchem State Node im Workflow der Request ausgeht und welche Transition im Workflow beschritten werden soll.

Im Normalfall stößt die Bearbeitung des Requests nicht auf Fehler. Die Eingabedaten können Business Rules auslösen. Dann wird die Transition auf den nachfolgenden State Node oder Decision Node verfolgt, wobei die mit den beteiligten Workflow-Elementen verknüpften Operationen ausgeführt werden. Zeigt die Transition direkt auf einen State Node, werden dessen Ausgabedaten zur Ausgabe verwendet und der Workflow auf diesen State Node weitergeschaltet. Liegt ein Decision Node vor, wird anhand der Decision Operation eine weitere Transition auf einen nachfolgenden State Node verfolgt. Mit diesem wird ansonsten wie oben verfahren.

Werden bei der Bearbeitung des Requests Benutzerfehler festgestellt, werden die Eingabedaten automatisch um Fehlermeldungen angereichert und die Ausgabe anhand des ursprünglichen State Nodes mit den angereicherten Eingabedaten erzeugt. Damit ist es möglich, in einer Web-Anwendung ein fehlerhaft ausgefülltes Formular mit allen Benutzereingaben erneut darzustellen, wobei die fehlerhaften Eingabefelder markiert und mit Fehlermeldungen erläutert werden.

Ein Workflowmodell kann um spezielle Transitions oder State Nodes für kritische Fehler, Anwendungsfehler und Timeouts erweitert werden. Tritt während der Abarbeitung eine solche Ausnahmesituation auf, wird die ursprüngliche Transition nicht ausgeführt, sondern stattdessen auf dem aktuellen State Node eine Fehler- oder Timeout-Transition gesucht, die zu einem anwendungsspezifischen Fehler- bzw. Timeout-Zustand verweist. Existiert eine solche Transition nicht, wird ein spezieller Fehler- oder Timeout-State-Node gesucht und direkt ausgegeben. Wurde auch dieser nicht modelliert, wird ein fest definierter Fehlerzustand eingenommen.

4.3 Validierung und Datenübernahme

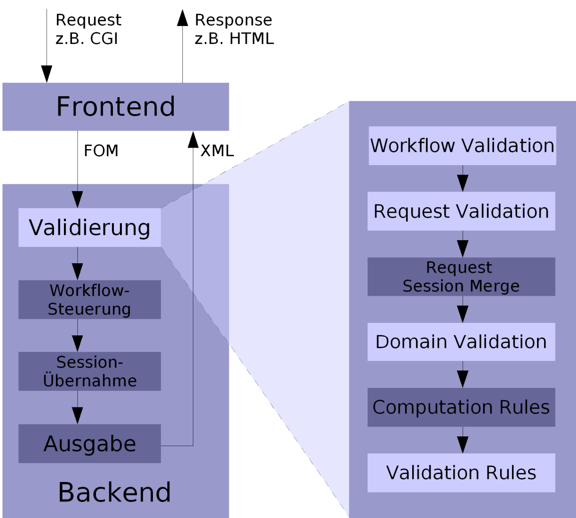

Abbildung 6: Schematischer Aufbau der Validierung

Vor der eigentlichen Bearbeitung eines Requests wurden die frontend-spezifisch formatierten Eingabedaten (Kontrollflußinformationen, modellierte und nicht-modellierte Anwendungsdaten) in die hierarchische XML-kompatible Darstellung des FOM-Modells verwandelt. Das Backend der *zen Engine* arbeitet danach immer auf dem FOM-Modell.

Die Übernahme externer Eingangsdaten in die Session ist in mehrfacher Hinsicht ein sicherheitskritischer Vorgang. Die Eingabedaten werden Bestandteil der Session, lösen die Workflow-Weiterschaltung aus, werden in Operationen weiterverarbeitet und sind später eventuell als Ausgabedaten wieder Bestandteil einer Response. Um die Datenkonsistenz und die korrekte Weiterverarbeitung sicherzustellen, werden die Eingabedaten einer mehrstufigen Validierung unterzogen.

Grundsätzlich wird bei Anwendungsfehlern und bei kritischen Fehlern die Bearbeitung eines Requests abgebrochen (Ausnahme: Transitionen, die mit einer *erroraware*-Action bezeichnet sind. Hier wird trotz Anwendungsfehlern die Bearbeitung fortgesetzt). Anstelle des normalen Ablaufs wird dann auf dem zuletzt verfügbaren State Node eine Transition gesucht, die mit einer Action namens *builtin:error* bezeichnet ist. Existiert so eine Transition, wird direkt in den damit verbundenen State Node gewechselt. Wenn nicht, wird spontan in einen ggf. vorhanden State Node namens *builtin:error* gewechselt. Existiert weder eine solche Transition noch ein solcher State Node, wird ein interner Fehlerzustand mit fest definierter Ausgabe eingenommen.

Wenn die Eingabedaten schließlich in der FOM-Darstellung ans Backend übergeben wurden, beginnt die integrierte Multi-Level-Validierung.

4.3.1 Workflow Validation (Level 1)

Zur Validierung wertet die *zen Engine* zuerst die Kontrollflußinformationen des Requests aus.

Fehler

Die *zen Engine* bricht die Bearbeitung sofort ab und nimmt für die aktuelle Session einen internen Fehlerzustand mit fest definierter Ausgabe (s. Kapitel 6.6.2 Fehlerzustand) ein, wenn:

- ein angegebener State Node nicht im Workflow existiert
- kein State Node angegeben wurde und kein State Node im Workflow als *defaultentry* gekennzeichnet ist

Die *zen Engine* betrachtet es als kritischen Fehler, wenn:

- ein State Node als Einsprungpunkt angegeben wird, der weder als *entry* noch als *defaultentry* gekennzeichnet ist
- keine Transition vom angegebenen State Node mit der angegebenen Action existiert
- keine Action angegeben wurde und vom angegebenen State Node mehr als eine Transition ausgeht (Ausname: Transitionen, die mit *builtin:error*- bzw. *builtin:timeout*-Actions bezeichnet sind werden dabei nicht mitgezählt)

Timeout

Wird bei Erhalt des Requests festgestellt, daß inzwischen ein Session-Timeout aufgetreten ist, wird auf dem im Request angegebenen State Node eine Transition gesucht, die mit einer Action namens *builtin:timeout* bezeichnet ist. Existiert so eine Transition, wird direkt in den so verbundenen State Node gewechselt. Existiert keine *builtin:timeout*-Transition, wird spontan in einen ggf. vorhandenen State Node namens *builtin:timeout* gewechselt. Existiert weder eine solche Transition noch ein solcher State Node, wird ein interner Fehlerzustand mit fest definierter Ausgabe eingenommen.

Konsistenzprüfung

Bei einer gültigen Session wird der im Request angegebene State Node mit dem zuletzt betretenen State Node verglichen, der in der Session gespeichert ist. Im Normalfall stimmen diese State Nodes überein und die Bearbeitung des Requests wird fortgesetzt. Ist das jedoch nicht der Fall, ist der Request inkonsistent zur Session. Falls eine *terminal*-Action angegeben wurde, wird der Request normal bearbeitet. Ist ansonsten der im Request angegebene State Node als *entry* oder *defaultentry* gekennzeichnet, wird eine neue Session begonnen, d.h. alte Sessiondaten werden gelöscht und der Request wird wie eine neue Anfrage bearbeitet. In allen andere Fällen liegt ein Konsistenzfehler vor (typischerweise durch Benutzung der Anwendung in zwei gleichzeitig geöffneten Browser-Fenstern bei Verwendung von Cookies, bei Nutzung von Browser-Funktionen wie Vorwärts/Zurück/Reload, o.ä.). Die Bearbeitung wird in diesem Fall abgebrochen, der Request verworfen und der zuletzt in der Session vorhandene State Node wird ausgegeben, um die Konsistenz zwischen Browser und Session wiederherzustellen.

Normalerweise folgt im Anschluß die nächste Validierungsstufe, sofern die Bearbeitung nicht aufgrund von Ausnahmefällen abgebrochen wurde.

4.3.2 Request Validation (Level 2 + 3)

Auf der Stufe der Request Validation werden die im Request enthaltenen, modellierten Anwendungsdaten anhand des Datenmodells überprüft.

Strukturvalidierung der modellierten Anwendungsdaten (Level 2)

Im Anwendungsmodell ist durch die Verknüpfung von Workflow und Datenmodell für jeden State Node festgelegt, welcher Ausschnitt aus dem Datenmodell in diesem State Node als Eingabe erwartet wird.

Bei der Zuordnung eines Datenelements zu einem State Node können im Modell bezüglich der Eingabe folgende Angaben gemacht werden:

- **In**: Ein Datenelement, das auf einem State Node als *in* markiert ist, muß Bestandteil der Eingabe sein.
- **In-Opt**: Ist das Datenelement auf diesem State Node als *in-opt* markiert, ist es optionaler Bestandteil der Eingabe, d.h. darf dabei sein, kann aber auch fehlen.

In der Strukturvalidierung wird nun erstens überprüft, ob alle Datenelemente, die im Datenmodell als *in* für den im Request angegebenen State Node markiert wurden, auch tatsächlich im Request enthalten sind. Zweitens wird untersucht, ob Datenelemente im Request enthalten sind, die weder als *in* noch als *in-opt* markiert wurden. Fehlen notwendige Datenelemente im Request oder sind überschüssige Datenelemente im Request vorhanden, wird dies als Anwendungsfehler gewertet, d.h. bei einer *erroraware*-Action werden solche Fehler gesammelt und die Bearbeitung fortgesetzt, bei allen anderen Actions wird die Bearbeitung wie oben beschrieben abgebrochen, da der Request nicht dem Anwendungsmodell entspricht.

Syntaktische Validierung (Level 3)

Nach der Strukturvalidierung folgt die syntaktische Validierung. Diese Validierungsstufe wird bei *cancel*- oder *clear*-Actions komplett übersprungen und bei *nonvalidating*-Actions nur partiell ausgeführt.

Da Requests an die *zen Engine* textbasierte HTML-Formular- oder XML-Daten enthalten, liegen die Inhalte von Atomen zunächst noch als Strings vor. Daher müssen sie syntaktisch validiert und anschließend, für die weitere Verarbeitung auf Objekt-Ebene, in die jeweils modellierten Datentypen konvertiert werden. Die syntaktische Validierung unterteilt sich in 3 Schritte:

- **Pflichtdatenvalidierung**: Wurden Atome als Pflichtdaten modelliert, dürfen ihre Inhalte nicht leer sein.
- **Längenvalidierung**: Die optional im Modell angegebene maximale Länge von Atomen darf nicht überschritten sein.
- **Typkorrektheit**: Jedem Atom wurde im Modell ein Datentyp zugeordnet. Das Atom enthält zunächst die Benutzereingabe als String, die unter Berücksichtigung der Länder- und Spracheinstellung (Locale) des Requests dem zugeordneten Datentyp entsprechen muß. Diese Stringrepräsentation wird gleichzeitig mit der Validierung in ein Objekt des entsprechenden Datentyps konvertiert.

Bei der syntaktischen Validierung festgestellte Fehler werden als Benutzerfehler gewertet. Die im Repository modellierten Fehlermeldungen für Benutzerfehler werden gesammelt, damit sie in der späteren Datenausgabe zur Verfügung stehen. Die Bearbeitung wird fortgesetzt, eventuell werden in der Folge jedoch Bearbeitungsschritte übersprungen.

Bei *nonvalidating-, cancel*- und *clear*-Actions werden alle Benutzerfehler ignoriert, die Requestdaten enthalten also niemals Fehlermeldungen. Bei *nonvalidating*-Actions werden nur korrekt formatierte Benutzereingaben in Objekte konvertiert, bei *cancel*- und *clear*-Actions erfolgt keine Konvertierung. Bei nicht konvertierten Atomen liegen die Dateninhalte also nach wie vor nur als Stringrepräsentationen vor.

4.3.3 Verschmelzung von Request- und Sessiondaten

Für die weitere Bearbeitung, insbesondere für die Ausführung von Operationen, wird nun eine Arbeitskopie der Daten erstellt (nachfolgend Arbeitsdaten genannt). Dieser Vorgang ist wieder abhängig vom aktuellen Action Type. Bei *cancel*-Actions enthalten diese Arbeitsdaten

lediglich eine Kopie der Sessiondaten, die Requestdaten werden ignoriert. Bei *clear*-Actions werden zusätzlich alle Dateninhalte von denjenigen Atomen aus der Session gelöscht, die im Request vorhanden waren.

Bei allen anderen Action Types werden Request- und Sessiondaten zu einer Gesamtsicht auf alle bisher angefallenen Daten verschmolzen. Dazu wird eine Kopie der Sessiondaten angelegt und alle Eingabedaten des Requests in diese Kopie integriert. Alte Sessiondaten werden dabei evtl. durch die aktuellen Eingabedaten des Requests überschrieben. Elemente unterhalb einer Liste werden gemäß ihrer Reihenfolge in Request und Session miteinander verschmolzen. Die sich so ergebende Listengröße entspricht dem Maximum der Größe von Request- und Sessionliste.

Sofern während der syntaktischen Validierung Benutzerfehler identifiziert wurden, sind die entsprechenden Fehlermeldungen nun auch in den Arbeitsdaten enthalten.

4.3.4 Domain Validation (Level 4)

Diese Validierungsstufe wird nur bei einer *default*- oder *terminal*-Action ausgeführt, sofern keine Benutzerfehler vorliegen, sowie bei einer *erroraware*-Action.

Jedem Atom kann im Datenmodell eine Wertemenge (Domäne) zugewiesen werden. In der Domain Validation wird überprüft, ob das Atom dann auch tatsächlich einen Wert aus der zulässigen Menge enthält. Anderenfalls wird dies als Benutzerfehler gewertet, dessen Fehlermeldung, wie bei jedem Benutzerfehler, den Arbeitsdaten hinzugefügt wird.

4.3.5 Computation Rules

Diese Bearbeitungsstufe wird nur bei einer *default*- oder *terminal*-Action ausgeführt, sofern keine Benutzerfehler vorliegen, sowie bei einer *erroraware*-Action.

Die Eingabedaten wurden inzwischen auf Ebene der Einzeldaten komplett validiert. Computation Rules dienen dazu, den validierten Datenraum um zusätzliche berechnete Daten zu erweitern. Eine Computation Rule wird immer dann aufgerufen, wenn sich Daten geändert haben, die gleichzeitig als Eingabeparameter dieser Computation Rule definiert sind. Als geändert gelten folgende Daten:

- Beliebige Elemente eines Requests, die zuvor noch nicht in der Session enthalten waren
- Atome eines Requests, deren Dateninhalte sich gegenüber den Inhalten der Sessiondaten verändert haben
- Listen eines Requests, deren Anzahl an Kind-Elementen sich gegenüber der Session verändert hat
- Rückgabewerte von Computation Rules. Diese werden in die Arbeitsdaten übernommen und können so weitere Computation Rules auslösen

Die Computation Rules dienen somit nicht zur eigentlichen Datenvalidierung, aber als vorbereitende Maßnahme für die Ausführung von Validation Rules. Computation Rules können dennoch definierte Fehler auslösen (eine sog. *OperationException*), die als Benutzerfehler gewertet werden. Die im Repository modellierte Fehlermeldung des jeweiligen Benutzerfehlers wird dann den Arbeitsdaten hinzugefügt.

Tritt im Java-Code der Computation Rule ein undefinierter Fehler auf (z.B. ein nicht behandelter Laufzeitfehler), wird dies von der *zen Engine* dagegen als Anwendungsfehler gewertet. Bei einer *erroraware*-Action wird die Bearbeitung dann fortgesetzt und der Fehler aggregiert, bei einer *default*- oder *terminal*-Action wird der Fehler dagegen als kritischer Fehler gewertet und die Bearbeitung abgebrochen.

4.3.6 Validation Rules (Level 5)

Diese Validierungsstufe wird nur bei einer *default*- oder *terminal*-Action ausgeführt, sofern keine Benutzerfehler vorliegen, sowie bei einer *erroraware*-Action.

Die mit Level 4 abgeschlossene Basis-Validierung der Eingabedaten ist für die meisten Anwendungsfälle noch nicht ausreichend. Daher sorgen die Validation Rules nach der Ausführung der Computation Rules für den abschließenden 5. Level der Validierung. Sie validieren geschäftsspezifische übergeordnete Zusammenhänge auf den kompletten Arbeitsdaten.

Validation Rules werden wie Computation Rules datengetrieben ausgeführt. Sie werden aufgerufen, wenn sich Daten geändert haben, die als Eingabeparameter der Validation Rule definiert sind. Dies betrifft gegenüber der Session geänderte Eingabedaten eines Request, insbesondere aber auch Rückgabewerte einer Computation Rule.

Validation Rules haben als reine Validierungsstufe – anders als Computation Rules – keine Rückgabewerte. Sie können stattdessen definierte Fehler auslösen (eine sog. *ValidationRuleException*), die als Benutzerfehler gewertet werden. Die im Repository modellierte Fehlermeldung des Benutzerfehlers wird dann den Arbeitsdaten hinzugefügt.

Tritt im Java-Code der Validation Rule ein undefinierter Fehler auf (z.B. ein nicht behandelter Laufzeitfehler), wird dies als Anwendungsfehler gewertet und unterliegt der gleichen Behandlung wie bei Computation Rules.

Mit Abschluß der Validation Rules ist die Validierung der Eingabedaten beendet. Sofern nach der automatischen Ausführung aller relevanten Validation Rules keine Fehler vorliegen, ist garantiert, daß die aktuellen Arbeitsdaten bezüglich Datenmodell und Geschäftsregeln korrekt und konsistent sind.

4.4 Workflowsteuerung

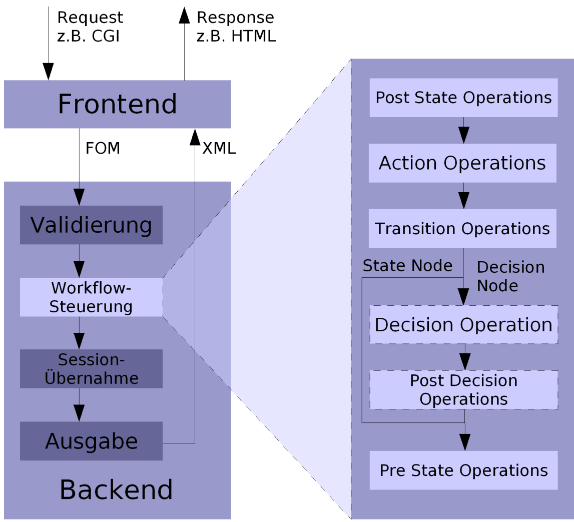

Abbildung 7: Schematischer Aufbau der Workflowsteuerung

Im Anschluß an die Validierung tritt die Workflowsteuerung in Aktion. Die Kontrollfluß-informationen des Requests wurden schon zu Beginn der Validierung überprüft und für gültig befunden, das heißt vom aktuellen State Node existiert eine mit der angegebenen Action belegte Transition zu einem nachfolgenden State Node oder Decision Node. Die Workflowsteuerung der *zen Engine* führt nun Schritt für Schritt den Zustandsübergang durch und ruft dabei alle modellierten Workflow Operations zum gewünschten Zeitpunkt auf.

Alle Workflow Operations können ebenso wie Computation und Validation Rules definierte Fehler auslösen (eine sog. *OperationException*), die als Benutzerfehler gewertet werden. Die im Repository modellierte Bezeichnung des Benutzerfehlers wird den Arbeitsdaten hinzugefügt.

Sobald ein Benutzerfehler vorliegt, wird die Workflowsteuerung in jedem Fall unterbrochen und der letzte Zustand wird ausgegeben, falls keine *erroraware*-Action verwendet wird.

Tritt im Java-Code einer Workflow Operation ein undefinierter Fehler auf (z.B. ein nicht behandelter Laufzeitfehler), wird dies dagegen als Anwendungsfehler gewertet. Bei einer *erroraware*-Action wird die Bearbeitung fortgesetzt und der Fehler aggregiert, bei allen anderen Action Types wird der unerwartete Fehler als kritischer Fehler gewertet und die Bearbeitung abgebrochen. Es wird ein interner Fehlerzustand mit fest definierter Ausgabe eingenommen.

Post State Operations

Während des Zustandsübergangs wird der zuletzt aktuelle State Node verlassen. Als erste Gruppe von Workflow Operations werden daher die Post State Operations des zuletzt aktuellen State Nodes aufgerufen.

Action Operations

Der Zustandsübergang wird von der im Request vorliegenden Action ausgelöst. Als nächste Gruppe von Workflow Operations werden die Action Operations dieser Action aufgerufen.

Transition Operations

Der Zustandsübergang zum nächsten State Node oder Decision Node wird durchgeführt. Als nächste Gruppe von Workflow Operations werden die Transition Operations der beschrittenen Transition aufgerufen.

Decision Operation

Ist das Ziel der Transition ein Decision Node, wird die Decision Operation aufgerufen. Diese gibt ein stringbasiertes Entscheidungskriterium zurück, das bestimmt, welche nachfolgende Transition zu einem State Node beschritten wird. Tritt bei Ausführung der Decision Operation ein definierter Benutzerfehler oder ein nicht behandelter Laufzeitfehler auf, wird kein nachfolgender State Node ausgewählt, stattdessen fällt der Workflow auf den zuletzt aktuellen State Node zurück.

Post Decision Operations

Diese Bearbeitungsstufe wird auch übersprungen, wenn die Decision Operation fehlgeschlagen ist, da in diesem Fall nicht klar ist, welcher Transition gefolgt werden soll.

Ein Decision Node liefert anhand seiner Decision Operation ein Entscheidungskriterium zurück, das die Transition vom Decision Node zum nachfolgenden State Node bestimmt. Als nächste Gruppe von Workflow Operations werden die an dieser Transition definierten Post Decision Operations ausgeführt.

Pre State Operations

Zum Abschluß des Zustandsübergangs wird ein State Node erreicht. Als letzte Gruppe von Workflow Operations werden die an diesem State Node definierten Pre State Operations aufgerufen.

4.5 Sessionübernahme

Wurde der Request ohne Fehler bearbeitet oder wurde eine *erroraware*-Action eingesetzt, werden die Arbeitsdaten in die Session geschrieben und ersetzen die zuvor in der Session gespeicherten Anwendungsdaten. Der Workflow wird auf den neuen State Node weitergeschaltet, die Kontrollflußinformationen werden bei allen Action Types außer bei *terminal*-Actions ebenfalls in der Session abgespeichert. Bei *terminal*-Actions hat aus Sicht des nächsten Requests in dieser Session kein State-Node-Wechsel stattgefunden, obwohl der Request komplett bearbeitet wurde. Die nachfolgende Ausgabe erfolgt jedoch bei allen Action Types auf Basis des neuen State Nodes.

Wurde keine *erroraware*-Action eingesetzt, werden bei fehlerhafter Bearbeitung des Request folgende Fälle unterschieden:

- Der Request konnte ohne kritischen Fehler bearbeitet werden, enthält jedoch Benutzerfehler: Die Arbeitsdaten werden nur zur Datenausgabe genutzt und anschließend verworfen. Die Sessiondaten werden dann nicht verändert. Die Ausgabe wird anhand des zuvor gültigen State Nodes durchgeführt, d.h. der tatsächliche Wechsel auf den neuen State Node kann effektiv noch von einer fehlerhaften Pre State Operation dieses neuen State Nodes blockiert werden.

- Die Bearbeitung wurde aufgrund eines kritischen Fehlers oder eines Anwendungsfehlers abgebrochen: Jetzt wird eine auf dem zuvor gültigen State Node definierte Transition gesucht, die mit einer *builtin:error*-Action bezeichnet ist. Existiert diese, wird die Transition zu dem nachfolgenden State Node beschritten. Wenn nicht, wird ein State Node names *builtin:error* gesucht. Wird auf die eine oder andere Art ein Fehler-State-Node gefunden, wird dieser in der Session als aktueller State Node gespeichert. Bis auf den State-Node-Wechsel bleiben die Sessiondaten unangetastet. Die Datenausgabe erfolgt anhand der Arbeitsdaten auf dem so bestimmten State Node. Existiert weder eine *builtin:error*-Transition noch ein *buitin:error*-State-Node, wird interner Fehlerzustand mit fest definierter Ausgabe

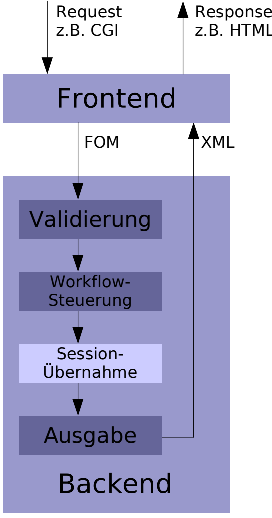

Abbildung 8: Sessionübernahme

eingenommen.

4.6 Datenausgabe

Die Ausgabe des Backends wird als XML-Text anhand des aktuellen State Nodes, des Datenmodells und der Session-Inhalte erzeugt. Die Ausgabe enthält

- alle Attribute und Ressourcen des ausgegebenen State Nodes
- alle Actions, die in den ausgehenden Transitionen des State Nodes verwendet werden (ebenfalls mit ihren Attributen und Ressourcen)
- alle dem State Node zur Ausgabe zugeordneten Daten-Elemente (mit ihren Attributen und Ressourcen)

Die Daten-Elemente können auf folgende Arten zugeordnet werden:

- **Out:** Daten-Elemente, die für einen State Node als *out* gekennzeichnet sind, werden anhand der Session bzw. des Datenmodells ausgegeben.
- **Out-Opt:** Daten-Elemente, die für einen State Node als *out-opt* gekennzeichnet sind, werden nur dann ausgegeben, wenn sie auch in der Session liegen.

Die Vorgehensweise ist wie folgt: Zuerst wird bestimmt, ob ein Element ausgegeben werden soll. Ist das der Fall, wird eine XML-Repräsentation des Elements, seiner Attribute und Ressourcen erzeugt. Ist das Element in der Session vorhanden, erfolgt die Ausgabe anhand der Sessioninformation. Insbesondere werden vorhandene Dateninhalte von Atomen ausgegeben. Dazu werden die Datentyp-Konvertierer der entsprechenden Datentypen benutzt, die eine sprachspezifische Stringrepräsentation des Objekt-Dateninhalts erzeugen. Ist das Element nicht in der Session vorhanden, erfolgt bei normaler *out*-Kennzeichnung die Ausgabe anhand des Datenmodells (hier sind keine Dateninhalte verfügbar). Bei *out-opt*-

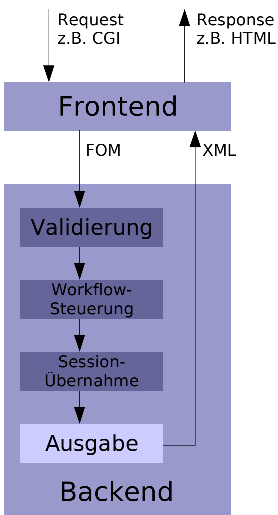

Abbildung 9: Datenausgabe

Kennzeichnung werden Elemente ausgelassen, wenn sie nicht in der Session vorhanden sind. Hat ein Atom eine zugeordnete Domäne, werden die Domänenwerte in die XML-Ausgabe aufgenommen. Bei Listen kann im Datenmodell eine Defaultgröße angegeben werden. Wird eine Liste anhand des Datenmodells ausgegeben, werden entsprechend viele Kind-Elemente in die XML-Ausgabe aufgenommen.

5 Anwendungsentwicklung mit dem *zen Developer*

Mit dem *zen Developer* können Anwendungen für die *zen Platform* mit Hilfe einer graphischen Oberfläche entworfen, entwickelt, getestet und gepflegt werden. Der *zen Developer* ist als Plugin für Eclipse realisiert und besteht aus einer Reihe von Komponenten, die nach Eclipse-Standard ein- und ausgeblendet werden können.

zen Perspective

Mit der *zen Perspective* können die Views des *zen Developer* in der üblichen Eclipse-Logik verwaltet werden. Die *zen Perspective* kann über *Window -> Customize Perspective...* den individuellen Bedürfnissen angepaßt werden bzw. über *Window -> Reset Perspective* auf den Auslieferungszustand zurückgesetzt werden.

Editor

Zentrales Element einer *zen*-Anwendung ist der

- **zen Editor**: Im *zen Editor* wird der Workflow einer Anwendung erstellt und bearbeitet.

Views

Zusätzlich existieren mehrere Views für den *zen Developer*, die jeweils einen Ausschnitt aus dem Anwendungsmodell anzeigen und editierbar machen:

- **Application**: Eigenschaften einer *zen*-Anwendung
- **Workflow**: Eigenschaften von State Nodes, Decision Nodes und Transitions
- **Action**: Definition von Actions
- **Data Model**: Definition des zentralen Datenmodells einer *zen*-Anwendung
- **Business Logic**: Definition der Geschäftslogik als Operations zur Einbindung von anwendungsspezifischem Java-Code in das Anwendungsmodell
- **Datatype**: Definition von Datentypen, die von Atoms im Datenmodell genutzt werden
- **Domain**: Definition von Wertebereichen (Domänen), die von Atoms im Datenmodell genutzt werden können
- **Resource**: Definition von sprachspezifischen Elemente (Texte, Pfade, etc.), die zur Ausgabe genutzt werden können. Ressourcen werden z.B. in einer Web-Anwendung für die Vertextung der Seiten oder für Fehlermeldungen eingesetzt.
- **Debug Data Model**: Während des Debugging-Vorgangs werden in der *Debug Data Model View* die Daten der aktuellen Session und die XML-Ausgabe angezeigt.

Zusätzlich werden einige Eclipse-Standard-Views für den *zen Developer* genutzt:

- **Outline**: Zeigt bei aktiviertem *zen Editor* eine Modellübersicht an
- **Console**: Wird für Logging-Output während des Debugging-Vorgangs genutzt
- **Tasks**: Enthält Fehlermeldungen bei Modellierungsfehlern

zen-Menü

Der *zen Developer* stellt im Menü *Zen* folgende Funktionen zur Verfügung:

- **Edit Repository**: Ruft den Dialog zur Verwaltung des Repository auf (s. Kapitel 5.2 Verwalten des Repository).
- **Reload Repository**: Lädt das Anwendungsmodell erneut aus der Datenbank und überschreibt (auf Nachfrage) alle lokalen, nicht gesicherten Änderungen.
- **Synchronize Running Application**: Aktualisiert eine laufende *zen*-Anwendung mit parallelen Änderungen im Anwendungsmodell.
- **Clear Output Cache**: Löscht den Ausgabe-Cache. Muß aufgerufen werden, um Änderungen an einem XSL Stylesheet für eine laufende *zen*-Anwendung zu übernehmen.
- **Mouseover Introspection**: Zeigt Breakpunkte auf Workflow-Elementen beim Überfahren mit dem Mauszeiger an, sofern aktiviert.
- **Details**: Zeigt Action-Namen und Return Values von Transitionen im *zen Editor* an, sofern aktiviert.
- **Auto Layout**: Ordnet die Workflow-Elemente automatisch neu an.
- **Zoom In/Out**: Zoomfunktion für den *zen Editor*
- **Create New Process**: Anlegen eines neuen Prozesses
- **Delete Open Process**: Löschen des aktuell geöffneten Prozesses

- **Rename Open Process**: Umbenennen des aktuell geöffneten Prozesses

Toolbar

Einige der oben erwähnten Funktionen sind auch über die Toolbar am oberen Rand des *zen Developer* zugänglich. Des weiteren befindet sich in der Toolbar ein Auswahlmenü, bestehend aus Sprache-Land-Kombinationen, für die

- **Entwicklungslocale**: Hier kann die Spracheinstellung für die Bearbeitung und Anzeige im *zen Developer* geändert werden. Dadurch wird die Eingabe von sprachabhängigen Beschriftungen, Bildpfaden oder Daten erleichtert. Diese Einstellung hat keine Auswirkung auf das Laufzeitverhalten der Anwendung.

5.1 Erstellen einer neuen Anwendung

Eine *zen*-Anwendung besteht aus dem Anwendungsmodell, das im Repository abgelegt wird, den Konfigurationsdateien, die unter anderem die Verbindungsdaten für das Repository und die Konfiguration weiterer Dienste enthalten, den Java-Klassen für die Geschäftslogik und dem *zen*-File, das lokale Einstellungen zur Anwendung, z.B. das Modell-Layout, speichert.

Die Vorgehensweise beim Anlegen einer neuen *zen*-Anwendung unterscheidet sich in Bezug auf die Konfiguration des Repositories: Entweder man setzt auf die mitgelieferte, filebasierte Testdatenbank *hsqldb* auf, oder man verwendet eine eigene Datenbank, die zusätzlich zu konfigurieren ist.

In jedem Fall wird die *zen*-Konfiguration und das *zen*-File innerhalb eines Java-Projekts erstellt.

Testdatenbank

Auf einem Java-Projekt wird der Wizard zur Anlage der *zen*-Anwendung über *File* -> *New* -> *zen Application* oder über das Kontextmenü des Java-Projekts aufgerufen. In dem Wizard müssen folgende Einstellungen vorgenommen werden:

- **Project**: Java-Projekt, in dem die Konfiguration bzw. das *zen*-File abgelegt wird
- ***zen* Configuration**: Die Konfiguration für die mitgelieferte Testdatenbank muß einmalig pro Java-Projekt erstellt werden. Falls *Missing* angezeigt wird, muß diese durch Betätigen des Buttons *Create* erzeugt werden. Danach wechselt die Anzeige auf *Present*. In dem Repository können dann beliebig viele Anwendungen erstellt werden.
- **File Name**: Name des *zen*-Files, in dem die neue *zen*-Application abgelegt werden soll.
- **Repository**: Die standardmäßig erstellte *zen*-Konfiguration enthält unter der Bezeichnung *zen.repository* die mitgelieferte filebasierte Testdatenbank *hsqldb.* Diese Datenbank ist nicht multiuserfähig.
- **Application Name**: Name der neuen *zen*-Anwendung, unter dem diese im Repository definiert wird. Bei Angabe einer bereits existierenden Anwendung wird lediglich ein neues *zen*-File erstellt, mit dem das bestehende Anwendungsmodell aus dem Repository geöffnet werden kann. Bei Angabe einer neuen Anwendung wird diese im Repository neu eingerichtet.

Eigene Datenbank

Bei Verwendung von eigenen Datenbanken muß die *zen*-Konfiguration selbst erstellt werden. Die Konfiguration der Testdatenbank kann hierzu als Vorlage dienen. In einer selbst erstellten *zen*-Konfiguration können jedoch beliebig viele Repositories angegeben werden. Die Details der *zen*-Konfiguration sind in Kapitel 8 (Konfiguration der zen Platform beschrieben.

Die selbst erstellten Konfigurationsdateien müssen in einem Ordner *conf* unterhalb des Projektverzeichnisses abgelegt werden. Der Ordner muß als *Source Folder* des Projekts angemeldet werden, indem auf dem Projekt über das Kontextmenü *Properties* gewählt wird, dort der Dialogbereich *Java Build Path* aktiviert wird und unter dem Reiter *Source* der Order mit *Add Folder...* hinzugefügt wird.

Danach müssen einmalig, wie in Kapitel 5.2 (Verwalten des Repository) beschrieben, die nötigen Datenbanktabellen im Repository erzeugt werden.

Das weitere Vorgehen ist analog zur Einrichtung einer *zen*-Anwendung in der Testdatenbank, nur ist die gerade erstellte *zen*-Konfiguration dann bereits vorhanden (*Present*) und es kann eines von ggf. mehreren selbst konfigurierten Repositories ausgewählt werden.

5.1.1 Anlegen eines neuen XSL Stylesheets

Über *File* -> *New* -> *XSL Stylesheets* oder das Kontextmenü auf einem Projekt wird der Wizard zum Anlegen eines neuen XSL Stylesheets aufgerufen. Dabei wird ein Stylesheet auf Basis des generischen Stylesheets *generic.xsl* erzeugt und dem Projekt hinzugefügt. Dieses kann dann beliebig angepaßt werden.

- **Project**: Java-Projekt, in dem das neue XSL Stylesheet erzeugt wird
- **Application**: Die *zen*-Anwendung, für die das neue XSL Stylesheet erzeugt wird
- **Stylesheet Name**: Name des XSL Stylesheets
- **Options**: Angabe, ob das neu erzeugte XSL Stylesheet als *Default Stylesheet* der ausgewählten *zen*-Anwendung eingetragen werden soll

5.1.2 Application View

In der *Application View* können die grundsätzlichen Eigenschaften einer *zen*-Anwendung eingestellt werden.

- **Name:** Name der Anwendung
- **Default Stylesheet:** Angabe eines Stylesheets, das für die gesamte Anwendung genutzt wird. Sollen verschiedene Stylesheets abhängig von Prozessen, State Nodes oder auch abhängig von der Anwendungslogik zum Einsatz kommen, kann das *Default Stylesheet* mittels eines *Stylesheet Selector-Interceptors* dynamisch ausgewählt werden.
- **Default Runtime Locale:** Auswahl einer Sprach-Land-Kombination, in der die Anwendung standardmäßig ablaufen soll. Für andere definierte Lokalisierungen muß die Anwendung dann mit speziellen Parametern aufgerufen werden. Das genaue Parameterformat ist im Kapitel 6.6.1 Eingabeformate beschrieben.
- **Include FOM Types:** Wird diese Eigenschaft aktiviert, wird bei der Ausgabe jedem Element dessen Element-Typ (*atom, comp, list*) als Attribut *builtin:type* angefügt. Kann bei Verwendung von Stylesheets zur Formular-Konstruktion benutzt werden. Muß aktiviert sein, wenn das mitgelieferte generische Stylesheet *generic.xsl* benutzt wird. Im Stylesheet *generic.xsl* wird der grundlegende Formularaufriß anhand der Element-Typen gesteuert.
- **Include Datatypes:** Wird diese Eigenschaft aktiviert, wird bei der Ausgabe jedes Atoms dessen Datentyp als vollqualifizierter Java-Klassenname (z.B. *java.lang.Integer*) als Attribut *builtin:datatype* angefügt. Kann bei Verwendung von Stylesheets zur Formular-Konstruktion benutzt werden. Muß aktiviert sein, wenn das mitgelieferte generische Stylesheet *generic.xsl* benutzt wird. Im Stylesheet *generic.xsl* wird der HTML-Inputtyp (Textfeld vs. Checkbox) u.a. anhand der Datentypen gesteuert.
- **Include Length:** Wird diese Eigenschaft aktiviert, wird bei der Ausgabe jedes Atoms, für das eine Längenbeschränkung angegeben wurde, die Länge als Attribut *builtin:length* angefügt. Kann bei Verwendung von Stylesheets zur Formular-Konstruktion benutzt werden. Im mitgelieferten generischen Stylesheet *generic.xsl* wird dadurch die Größe bzw. die maximale Eingabelänge von HTML-Eingabefeldern eingestellt.
- **Include Mandatory:** Wird diese Eigenschaft aktiviert, wird bei der Ausgabe jedes Atoms, das als *mandatory* definiert ist, das Attribut *builtin:mandatory="true"* angefügt. Kann bei Verwendung von Stylesheets zur Formular-Konstruktion benutzt werden. Im mitgelieferten generischen Stylesheet *generic.xsl* wird dadurch das zugehörige Label des Atoms (Ressource mit dem Namen *label*) mit einem Stern(*) gekennzeichnet.
- **Include Read Only:** Wird diese Eigenschaft aktiviert, wird bei der Ausgabe jedes Atoms, das auf dem auszugebenden State Node nicht als In-Element definiert ist, das Attribut *builtin:readonly="true"* angefügt. Kann bei Verwendung von Stylesheets zur Formular-Konstruktion benutzt werden. Muß aktiviert sein, wenn das mitgelieferte generische Stylesheet *generic.xsl* benutzt wird. Im Stylesheet *generic.xsl* werden Atome, die so gekennzeichnet sind, nicht als HTML-Eingabeelement, sondern als bloßer Text bzw. Grafik (Checkboxen) gezeichnet.

5.2 Verwalten des Repository

Über das Menü *Zen -> Edit Repository...* wird ein Dialog zum Verwalten von Repositories aufgerufen. Voraussetzung ist, daß bereits die *zen*-Konfiguration eingerichtet wurde und die verwendeten Datenbanken bereitgestellt sind. Im linken Fenster befinden sich alle im Workspace vorhandenen Projekte.

Im mittleren Fenster werden alle im ausgewählten Projekt konfigurierten Repositories angezeigt. Falls ein ausgewähltes Repository noch nicht eingerichtet wurde, ist der Button *Create Table Set* aktiviert. Mit Druck auf den Button werden alle nötigen Datenbank-Tabellen im Repository angelegt, danach ist der Button inaktiv.

Für ein ausgewähltes Repository werden im rechten Fenster alle dort enthaltenen *zen*-Anwendungen angezeigt. Hier kann eine *zen*-Anwendung mit *Delete Application* komplett und unwiederbringlich gelöscht werden. Mit *Create Application* wird der Wizard zum Anlegen einer neuen *zen*-Anwendung gestartet.

5.3 Öffnen und Bearbeiten einer Anwendung

Durch das Öffnen eines *zen*-Files wird die zugeordnete *zen*-Anwendung geöffnet. Der *zen Editor,* das zentrale Werkzeug zum graphischen Erstellen von Workflows, wird gestartet und alle Views des *zen Developer* mit den Inhalten der Anwendung gefüllt. Es lassen sich beliebig viele Anwendungen gleichzeitig öffnen, in den Views werden jeweils die Daten derjenigen Anwendung angezeigt, deren Editor aktiv ist. Wird eine Modifikation an einer Anwendung per Editor oder View vorgenommen, wird ihr Karteireiter über dem Editor mit einem Stern gekennzeichnet. Alle Änderungen einer Anwendung lassen sich über das Menü *Zen -> Reload Repository* rückgängig machen. Einzelne Änderungen können mit *Undo/Redo* bis zum Zeitpunkt des letzten Speicherns rückgängig gemacht bzw. erneut durchgeführt werden.

Ist die Anwendung mit einer Fehlermarkierung gekennzeichnet, liegt im Anwendungsmodell bzw. im Zusammenspiel von Anwendungsmodell und benutzerdefiniertem Java-Code ein schwerwiegender Fehler vor. Die *zen*-Anwendung ist dann möglicherweise nicht ablauffähig. In der *Task View* (Standard-View von Eclipse) ist der genaue Fehler mit seiner Beschreibung

zu finden. Mit Klick auf den Fehler wird diejenige View geöffnet, mit der der Fehler beseitigt werden kann. Die Anwendung kann auch mit einem Warnungssymbol gekennzeichnet sein. Dann ist die *zen*-Anwendung möglicherweise nicht in allen Bereichen korrekt ablauffähig. Die genauen Ursachen sind ebenfalls in der *Task View* zu finden. Auch Warnungen sollten ernst genommen und korrigiert werden.

Werden identische Teile einer Anwendung gleichzeitig von mehreren Entwicklern bearbeitet, kann es zu Konflikten kommen, die aufgelöst werden müssen. Will ein Entwickler eine Anwendung speichern, in der währenddessen ein anderer Entwickler identische Teile geändert und gespeichert hat, bekommt er eine Dialogbox, in der die betroffenen Anwendungselemente aufgelistet sind. Der Entwickler hat dann die Möglichkeit, die Änderungen der entsprechenden Elemente zu übernehmen und die Anwendung anschließend zu speichern.

5.4 Erstellen eines Workflows

Im *zen Editor* werden State Nodes und Decision Nodes angelegt und mit Transitions verbunden. Der Editor enthält eine Werkzeugpalette, in der die Workflow-Elemente ausgewählt werden. Diese können dann in der Zeichenfläche angelegt werden. Alle wichtigen Eigenschaften der Workflow-Elemente werden in der *Workflow View* definiert.

In der *Outline View* (eine Standard-View von Eclipse) wird der Workflow des *zen Editors* miniaturisiert dargestellt. Der im Editor sichtbare Ausschnitt ist in der *Outline View* farblich hervorgehoben und kann mit der Maus verschoben werden. Dabei wird der Inhalt des *zen Editors* mitverschoben.

Unter Umständen sind Nodes mit Fehler- oder Warnungssymbolen gekennzeichnet. Die genauen Beschreibungen sind in der *Task View* zu finden bzw. erscheinen als Hinweistext, wenn die Maus über die Symbole gehalten wird. Fehler müssen unbedingt gelöst werden, bevor die *zen*-Anwendung gestartet wird. Auch Warnungen sollten korrigiert werden.

Prozesse

An der unteren Kante des Editors sind Karteireiter angebracht, in denen der gesamte Workflow *(Overview)* bzw. Teilprozesse des Workflows (anhand der Prozeßnamen) angezeigt werden. Prozesse dienen insbesondere dazu, komplexe Workflows zu gliedern. Beim Anlegen einer neuen Anwendung wird automatisch ein Default-Prozeß (*default)* angelegt. Nodes werden denjenigen Prozessen zugeordnet, in denen sie angelegt werden. Nodes, die in der Übersicht (*Overview*) angelegt werden, werden dem Default-Prozeß zugeordnet. Zusätzliche Prozesse werden über das Menü *Zen* **->** *Create New Process* angelegt. Wird ein zusätzlich angelegter Prozess in den Vordergrund gebracht, kann er über das Menü *Zen* mit *Rename Open Process* umbenannt und mit *Delete Open Process* gelöscht werden. Nodes können über Prozeßgrenzen hinweg mit Transitions verbunden werden. Die *Workflow View* erlaubt die nachträgliche Zuordnung von Nodes zu bestimmten Prozessen.

Nodes und Transitions

Beim Anlegen eines Workflows muß ein State Node als Einsprungpunkt für den initialen Request existieren, der für den Aufruf einer Anwendung benötigt wird. Dieser wird mit genau einer Transition mit demjenigen State Node verbunden, der zur Erzeugung der initialen Response dient. Der initiale State Node kann über die *Data Model View* mit Aufrufparametern (Eingabedaten) versehen werden, die beim Aufruf der Anwendung mitübergeben werden können oder müssen. In der Regel besitzt der initiale State Node jedoch keine Visualisierung, so daß er nicht mit Ausgabedaten verknüpft wird. Der initiale State Node wird in der *Workflow View* mit der *Gate*-Eigenschaft *defaultentry* gekennzeichnet. Zusätzlich können weitere State Nodes mit der *Gate*-Eigenschaft *entry* gekennzeichnet werden, die alternative Einsprungpunkte in den Workflow definieren.

Transitionen können State Nodes mit anderen State Nodes oder mit Decision Nodes verbinden. Eine ausgehende Transition eines State Nodes kann auch reflexiv den gleichen State Node als Ziel haben. Jeder Transition, die von einem State Node ausgeht, wird in der *Workflow View* eine Action zugeordnet. Dabei müssen alle ausgehenden Transitionen des gleichen State Nodes mit einer unterschiedlichen Action bezeichnet sei, um sie unterscheiden zu können.

Eine Transition, die von einem Decision Node ausgeht, kann nur einen State Node als Ziel haben. Eine solche Transition wird nicht mit einer Action bezeichnet, sondern mit einem string-basierten Entscheidungskriterium. Einem Decision Node wird eine Decision Operation zugeordnet, die diese Entscheidungskriterien berechnet und zurückliefert. Für alle Kriterien, die eine Decision Operation berechnen kann, muß eine eigene Transition vorgesehen werden.

Nodes und Transitionen können mit der *Entfernen*-Taste (Entf/Del) gelöscht werden. Wird ein Node gelöscht, werden alle mit dem Node verbundenen Transitionen mitgelöscht.

builtin: Spezielle State Nodes

Für Anwendungsfehler, kritische Fehler und Timeouts können spezielle State Nodes verwendet werden. Diese bekommen die Namen *builtin:error* bzw. *builtin:timeout*. Soll die Behandlung von Fehlern und Timeouts feingranularer erfolgen, können Transitionen von State Nodes auf anwendungsspezifische Fehler- und Timeout-Zustände gezogen werden. Diese Transitionen müssen dann mit Actions namens *builtin:error* bzw. *builtin:timeout* bezeichnet werden.

Gibt es innerhalb der Anwendung keine globalen *builtin:error*- bzw. *builtin:timeout*-State-Nodes und existieren auf dem State Node, auf dem ein Fehler bzw. Timeout aufgetreten ist, keine *builtin:error*- bzw. *builtin:timeout*-Transitionen, wird von der *zen Engine* bei der Ausführung ein interner Fehlerzustand mit fest definierter Ausgabe eingenommen.

Debugging

Über das Menü *Zen -> Mouseover Introspection* kann die Detailansicht von Workflow-Elementen aktiviert werden. Dadurch können auf Workflow-Elementen graphische Breakpunkte für das Debugging gesetzt und gelöscht werden. Fährt man mit der Maus über ein Workflow-Element, wird es bei aktivierter *Introspection* vergrößert und zeigt alle möglichen Breakpunkte an, die dann per Doppelklick ein- und ausgeschaltet werden können. Ein Workflow-Element mit aktiviertem Breakpunkt wird blau gezeichnet.

5.4.1 Workflow View

Die *Workflow View* zeigt jeweils die Eigenschaften des im *zen Editor* markierten Workflow-Elements der aktiven Anwendung an. Die View zeigt Je nach Workflow-Element spezialisierte Eigenschaften an.

State Node

Wird ein State Node selektiert, lassen sich in der *Workflow View* folgende Eigenschaften einstellen:

- **Name**: Name des State Nodes. Muß mit Buchstaben anfangen und darf nur Buchstaben, Zahlen, Bindestriche (-), Unterstriche (_) und Punkte (.) enthalten.
- **Process**: Zuordnung des State Nodes zu einem Prozeß. Beim Wechsel zu einem anderen Prozeß wird dieser aus der aktuellen Prozeßansicht des Editors entfernt und in der Ansicht des anderen Prozesses angezeigt. Transitionen bleiben verbunden. Transitionen zwischen Nodes in verschiedenen Prozessen werden graphisch mit einem freien Pfeilende angezeigt.
- **Gate**: Bestimmt Ein- und Aussprungspunkte im Workflow:
  - *default*: Normaler State Node im Workflow. Ein Workflow kann nicht mit einem so gekennzeichneten State Node begonnen werden.
  - *entry*: Einsprungpunkt in den Workflow. Erlaubt den Einsprung in den Workflow an dieser Stelle.
  - *defaultentry*: Zentraler standardmäßiger Einsprungpunkt in den Workflow. Wird die Anwendung ohne Angabe eines State Nodes aufgerufen, wird der Workflow mit dem so gekennzeichneten State Node begonnen. Nur ein einziger State Node im Workflow darf so gekennzeichnet sein.
  - *exit*: Aussprungpunkt aus dem Workflow. Befindet sich ein Client auf einem so gekennzeichneten State Node, wird dessen Workflow als abgeschlossen betrachtet, was zu einer schnelleren Session-Löschung führen kann und somit Wartungsarbeiten vereinfacht.
- **Attributes**: Erlaubt es, Attribute für den State Node zu definieren. Die Attribute werden der XML-Ausgabe der Backend-Frontend-Schnittstelle hinzugefügt. Attributnamen müssen mit Buchstaben anfangen und dürfen nur Buchstaben, Zahlen, Bindestriche (-), Unterstriche (_) und Punkte (.) enthalten. Attribute werden verwendet, um dem Frontend sprachunabhängige technische Informationen mitzuteilen, die z.B. vom XSL-Stylesheet-Ersteller zur Visualisierung verwendet werden können.
- **Workflow Operations**: Einem State Node können *Pre State*- oder *Post State Operations* zugeordnet werden. Im Fenster „Available Workflow Operations“ werden alle in der *Business Logic View* definierten Workflow Operations angezeigt. Diese können mit dem Pfeil nach rechts dem State Node zugewiesen bzw. mit dem Pfeil nach links wieder entfernt werden. Mit den Pfeilen nach oben/unten kann die Reihenfolge verändert werden, wenn mehrere Workflow Operations benötigt werden. Im Fenster „Available Workflow Operations“ kann mit der rechten Maustaste über das Kontextmenü direkt in die entsprechende Java-Klasse bzw. Java-Methode gesprungen werden.
- **Resources**: Einem State Node können State-Resources zugeordnet werden. Im Fenster „Available Resources“ werden alle in der *Resource View* definierten State-Resources angezeigt. Diese können mit dem Pfeil nach rechts dem State Node zugewiesen bzw. mit dem Pfeil nach links wieder entfernt werden.

Decision Node

Wird ein Decision Node selektiert, lassen sich in der *Workflow View* folgende Eigenschaften einstellen:

- **Name**: Name des Decision Nodes. Muß mit Buchstaben anfangen und darf nur Buchstaben, Zahlen, Bindestriche (-), Unterstriche (_) und Punkte (.) enthalten.
- **Process**: Der Decision Node kann einem anderen Prozeß zugeordnet werden. Er wird aus der aktuellen Prozeßansicht des Editors entfernt und in der Ansicht des anderen Prozesses angezeigt. Transitionen bleiben verbunden. Transitionen zwischen Nodes in verschiedenen Prozessen werden graphisch mit einem freien Pfeilende angezeigt.
- **Decision Operation**: Einem Decision Node muß genau eine *Decision Operation* zugeordnet werden. Im Fenster werden alle in der *Business Logic View* definierten Decision Operations angezeigt. Die zu verwendende Decision Operation wird selektiert.

Ausgehende Transition eines State Nodes

Wird eine ausgehende Transition eines State Nodes selektiert, lassen sich in der *Workflow View* folgende Eigenschaften einstellen:

- **Action**: Der Transition wird eine in der *Action View* definierte Action zugeordnet
- **Operations**: Einer Transition können *Transition Operations* zugeordnet werden. Im Fenster „Available Transition Operations" werden alle in der *Business Logic View* definierten Workflow Operations angezeigt. Diese können mit dem Pfeil nach rechts der Transition zugewiesen bzw. mit dem Pfeil nach links wieder entfernt werden. Mit den Pfeilen nach oben/unten kann die Reihenfolge verändert werden, wenn mehrere Workflow Operations für die Transition benötigt werden.

Ausgehende Transition eines Decision Nodes

Wird eine ausgehende Transition eines Decision Nodes selektiert, lassen sich in der *Workflow View* folgende Eigenschaften einstellen:

- **Return Value**: Der Transition wird ein String zugeordnet. Dieser muß einem Entscheidungskriterium entsprechen, das die Decision Operation des Decision Nodes zurückliefern kann.
- **Operations**: Einer Transition können *Post Decision Operations* zugeordnet werden. Im Fenster „Available Post Decision Operations" werden alle in der *Business Logic View* definierten Workflow Operations angezeigt. Diese können mit dem Pfeil nach rechts der Transition zugewiesen bzw. mit dem Pfeil nach links wieder entfernt werden. Mit den Pfeilen nach oben/unten kann die Reihenfolge verändert werden, wenn mehrere Workflow Operations für die Transition benötigt werden

5.4.2 Action View

In der *Action View* werden Actions definiert, die in der *Workflow View* einzelnen Transitionen zugeordnet werden können. Actions werden mit der rechten Maustaste über den entsprechenden Menüeintrag angelegt und gelöscht. Sobald eine Action im Workflow verwendet wird, erhält sie ein Schloßsymbol und kann nicht mehr gelöscht werden.

- **Name**: Name der Action. Muß mit Buchstaben anfangen und darf nur Buchstaben, Zahlen, Bindestriche (-), Unterstriche (_) und Punkte (.) enthalten.
- **Type**: Bestimmt den Action Type und damit das Ablaufverhalten von Transitions:
    - *default*: Eine Transition mit einer *default*-Action ist der Normalfall. Andere Action Types werden nur in Ausnahmefällen gebraucht.
    - *cancel*: Bei einer Transition mit einer *cancel*-Action werden die Benutzereingaben weder validiert noch in die Session übernommen.
    - *clear*: Bei einer Transition mit einer *clear*-Action werden die Benutzereingaben nicht validiert und anschließend aus der Session gelöscht.
    - *nonvalidating*: Bei einer Transition mit einer *nonvalidating*-Action werden Benutzereingaben nicht validiert, jedoch in die Session übernommen.
    - *erroraware*: Eine Transition mit einer *erroraware*-Action bricht bei Anwendungsfehlern die Bearbeitung nicht ab. Die Fehler werden stattdessen aggregiert. Auch bei Benutzerfehlern werden alle Bearbeitungsschritte weiter ausgeführt.
    - *terminal*: Eine Transition mit einer *terminal*-Action kann parallel zum aktuellen Workflow laufen, Start- und Endzustände einer solchen Transition werden nicht auf Workflowkonsistenz geprüft.
- **Reiter Resources**: Einer Action können Action-Resources zugeordnet werden. Im Fenster „Available Resources" werden alle in der *Resource View* definierten Action-Resources angezeigt. Diese können mit dem Pfeil nach rechts der Action zugewiesen bzw. mit dem Pfeil nach links wieder entfernt werden.
- **Reiter Attributes**: Erlaubt es, Attribute für die Action zu definieren. Die Attribute werden der XML-Ausgabe der Backend-Frontend-Schnittstelle hinzugefügt. Attributnamen müssen mit Buchstaben anfangen und dürfen nur Buchstaben, Zahlen, Bindestriche (-), Unterstriche (_) und Punkte (.) enthalten. Attribute werden verwendet, um dem Frontend sprachunabhängige technische Informationen mitzuteilen, die z.B. vom XSL-Stylesheet-Ersteller zur Visualisierung verwendet werden können.
- **Reiter Operations**: Einer Action können Action Operations zugeordnet werden. Im Fenster „Available Action Operations" werden alle in der *Business Logic View* definierten Workflow Operations angezeigt. Diese können mit dem Pfeil nach rechts dem State Node zugewiesen bzw. mit dem Pfeil nach links wieder entfernt werden. Mit den Pfeilen nach oben/unten kann die Reihenfolge verändert werden, wenn mehrere Workflow Operations für diese Action benötigt werden. Im Fenster „Available Action Operations" kann mit der rechten Maustaste direkt in die entsprechende Java-Klasse bzw. Java-Methode gesprungen werden.

5.5 Erstellen eines Datenmodells

Das Datenmodell einer *zen*-Anwendung ist ein hierarchisches Modell, das in der *Data Model View* in einer Baumansicht erstellt wird. Es enthält alle Daten, die innerhalb der Anwendung zur Ein- und Ausgabe und als Parameter von Operationen verwendet werden. In der Regel ist das so definierte Modell das vollständige Datenmodell einer Anwendung. Innerhalb von Operationen kann natürlich trotzdem auf beliebige Java-Objekte zugegriffen werden.

Das Wurzelelement eines *zen*-Datenmodells ist eine fest definierte Composition namens *data*. Diese Composition wird in jedem State Node ausgegeben. Der Composition *data* können z.B. Ressourcen und Attribute zugeordnet werden, die unabhängig vom State Node immer ausgegeben werden sollen. An dieser Wurzel kann ein beliebiger Baum konstruiert werden.

5.5.1 Data Model View

In der Baumansicht können mit der rechten Maustaste bei Auswahl des entsprechenden Menüeintrags weitere Datenelemente hinzugefügt werden. Auf einer Composition kann eine weitere Composition, eine List oder ein Atom als Kind-Element angelegt werden. Auf einer List kann eine Composition oder ein Atom als Kind-Element angelegt werden (das Kind-Element bestimmt die Definition der Liste, zur Laufzeit kann eine Liste beliebig viele Kinder vom definierten Typ enthalten). Ebenfalls mit der rechten Maustaste und Auswahl des entsprechenden Menüeintrags bzw. mit der Entfernen-Taste kann jedes Element bis auf die *data*-Composition wieder gelöscht werden. Das Löschen von Elementen ist rekursiv, d.h. alle darunterliegenden Elemente werden mitgelöscht.

Unter Umständen können Elemente mit Fehlersymbolen gekennzeichnet sein. Die genauen Beschreibungen sind dann in der *Task View* zu finden bzw. erscheinen als Hinweistext in der Statuszeile. Fehler müssen unbedingt gelöst werden, bevor die *zen*-Anwendung gestartet wird.

Reorganisation

Bei der Ausgabe wird die Reihenfolge von Kind-Elementen einer Composition berücksichtigt. Einzelnen Datenelemente oder Teilbäume verschiebt man (abhängig vom Window-System), mit entsprechend gedrückter Maustaste an die gewünschte Stelle und läßt dort die Maustaste wieder los. In der Statuszeile wird während des Verschiebens die Pfadangabe des jeweiligen Ziels unter der Maus angezeigt. Falls ein Element nicht an die gewünschte Stelle verschoben werden kann, wird der Grund dafür ebenso in der Statuszeile angezeigt. Jedes Kind-Element kann innerhalb einer Composition kann auch über das Kontextmenü nach oben bzw. nach unten verschoben werden.

Werden Datenelemente verschoben, die in der Geschäftslogik verwendet werden, wird sicherheitshalber nachgefragt, ob das Verschieben tatsächlich durchgeführt werden soll. Da das Verschieben von referenzierten Datenelementen die urspüngliche Semantik der Geschäftslogik verändern kann, wird standardmäßig ein *TOCHECK*-Task in der *Task View* angelegt, der die (auch indirekt) beteiligten Operations beschreibt. Auf diesem Task werden über den Befehl *TOCHECK Details* genaueren Informationen angezeigt, die editierbar sind und somit Schritt-für-Schritt überprüft werden können.

Falls die *TOCHECK*-Tasks in der *Task View* nicht sichtbar sind, ist eventuell deren Task-Filter aktiviert. In diesem Fall muß der Filtertyp *zen Platform Indications* im Filter-Dialog wieder auf A*nzeigen* gesetzt werden.

Ein-/Ausgabe

Elemente aus dem Datenmodell können einzelnen State Nodes als *In* und *Out* zugeordnet werden. Damit wird definiert, welche Daten auf einem State Node als Eingabe (*In*) erwartet werden bzw. welche Daten ausgegeben werden (*Out*). Die Zuordnung wird in der Tabelle neben dem Baum in den Spalten *In/In-Opt/Out/Out-Opt* definiert. Ist im *zen Editor* kein State Node selektiert, ist die Belegung undefiniert. Bei selektiertem State Node werden die Spalteninhalte als editierbare und nicht-editierbare Checkboxen angezeigt. Die einzelnen Spalten bedeuten:

- *In*: Ein Datenelement, das auf einem State Node als *In* markiert ist, ist Bestandteil der Eingabe. Wird ein Element als *In* markiert, werden alle Kinder und Eltern automatisch ebenfalls als *In* markiert. Editierbar ist jedoch nur die Eigenschaft auf dem ursprünglichen Element, nicht auf den automatisch mitmarkierten Elementen.

- *In-Opt*: Ein *In*-Datenelement muß im selektierten State Node immer als Bestandteil der Eingabe vorliegen, außer es wird als *In-Opt* markiert. Dann ist es nur optionaler Bestandteil der Eingabe. Werden alle Kinder einer Composition bzw. das Kind einer Liste als *In-Opt* markiert, wird die Composition bzw. die Liste automatisch ebenfalls als *In-Opt* markiert. Editierbar ist jedoch nur die Eigenschaft auf den ursprünglichen Elementen, nicht auf den automatisch mitmarkierten Elemeten.

- *Out:* Elemente, die für einen State Node als *Out* gekennzeichnet sind, sind Bestandteil der Ausgabe. Wird ein Element als *Out* markiert, werden alle Kinder und Eltern automatisch ebenfalls als *Out* markiert. Editierbar ist jedoch nur die Eigenschaft auf dem ursprünglichen Element, nicht auf den automatisch mitmarkierten Elemeten.

- *Out-Opt:* Elemente, die für einen State Node als *Out-Opt* gekennzeichnet sind, werden nur dann ausgegeben, wenn sie auch in der Session liegen. Ist ein Element nicht Opt-Out und liegt nicht in der Session, wird es mit leerem Inhalt anhand des Datenmodells ausgegeben.

Nur wenn auf einem State Node Datenelemente nicht jedesmal als Ein- bzw. Ausgabe auftreten, sollte man *In-Opt* und *Out-Opt* verwenden. Das ist z.B. dann der Fall, wenn auf einem Eingabeformular ganze Teile abhängig von Geschäftslogik aus- und eingeblendet

werden sollen. In solchen Fällen ist *In-Opt* und *Out-Opt* in der Regel synchron zu sehen: Wird ein Element (fallweise) nicht ausgegeben, taucht es im nächsten Request in der Regel auch nicht als Eingabeelement auf (es sei denn, das Stylesheet erzwingt dies).

Der Dialogteil neben der Baumansicht ist je nach Datenelement spezialisiert. Er enthält jedoch auch allgemeine Eigenschaften.

Allgemein

Alle Datenelemente besitzen folgende Eigenschaften:

- **Resources**: Einem Element können Element-Resources zugeordnet werden. Im Fenster „Available Resources" werden alle in der *Resource View* definierten Element-Resources angezeigt. Diese können mit dem Pfeil nach rechts dem Element zugewiesen bzw. mit dem Pfeil nach links wieder entfernt werden

- **Attributes**: Erlaubt es, Attribute für das Element zu definieren. Die Attribute werden der XML-Ausgabe der Backend-Frontend-Schnittstelle hinzugefügt. Attributnamen müssen mit Buchstaben anfangen und dürfen nur Buchstaben, Zahlen, Bindestriche (-), Unterstriche (_) und Punkte (.) enthalten. Attribute werden verwendet, um dem Frontend sprachunabhängige technische Informationen mitzuteilen, die z.B. vom XSL-Stylesheet-Ersteller zur Visualisierung verwendet werden können

Atom

Ein Atom hat folgende zusätzliche Eigenschaften:

- **Datatype**: Jedem Atom wird ein Datentyp zugeordnet. Diese werden in der *Datatype View* definiert.

- **Max. Value Length**: Beschränkt die Länge der String-Repräsentationen von Atom-Inhalten. Wird eine Länge angegeben, muß eine Fehlermeldung (Data Error) ausgewählt werden, die bei Längenüberschreitung als Benutzerfehler verwendet wird. Die angegebene Länge wird bei der Ausgabe an das Stylesheet übermittelt und kann z.B. zur Längenbeschränkung von Eingabefeldern in Formularen verwendet werden.

- **Mandatory**: Ein so markiertes Atom muß bei der Eingabe immer mit einem Inhalt belegt werden. Zusätzlich wird eine Fehlermeldung (Data Error) ausgewählt, die bei leeren Inhalten als Benutzerfehler verwendet wird. Dient z.B. zur Behandlung von Pflichtfeldern in Formularen.

- **Default Value**: Einem Atom, das als *mandatory* definiert ist, kann ein zum Datentyp passender Default-Inhalt im Format der Default-Locale zugeordnet werden. Existiert das Atom nicht in der Session bzw. hat keinen Inhalt, wird der hier angegebene Inhalt bei der Ausgabe im Format der Session-Locale ausgegeben. So können z.B. Formular-Felder vorbelegt werden.

- **Key Element**: Eigenschaft für Atome, die entweder direkt unter einer List hängen oder als Kind einer Composition definiert sind, die direkt unter einer List hängt. Besitzt im ersten Fall ein so markiertes Atom keinen Inhalt, wird es von der *zen Engine* bei Bearbeitung eines Request aus der List gelöscht. Besitzen im zweiten Fall alle so markierten Atome der Composition keinen Inhalt, wird die Composition aus der List gelöscht. Das Löschen geschieht unabhängig davon, ob das Atom bei markierter *mandatory*-Eigenschaft leer war.
Verwendung findet diese Eigenschaft z.B. für Formulare, in denen Daten in Tabellenform einzugeben sind: Die List repräsentiert Tabellenzeilen, eine Composition bestimmt den Spaltenaufriß einer Zeile, die Atome sind die einzelnen Tabellenzellen. Eine Tabellenzeile, in der alle als *key element* markierten Zellen leer sind, wird komplett verworfen, auch wenn nicht ausgefüllte Zellen als *mandatory* definiert sind. In die Session werden nur Zeilen aufgenommen, bei denen alle *key-element*-Atome ausgefüllt sind. Als normale Verletzung der *mandatory*-Eigenschaft (und somit als Benutzerfehler) gilt es, wenn einige *key-element*-Atome einer Zeile ausgefüllt sind, andere *key-element*-Atome der gleichen Zeile nicht.

- **Domain**: Einem Atom kann eine Domäne (Wertemenge), die in der *Domain View* definiert wurde, zugeordnet werden. Die Atom-Inhalte müssen bei der syntaktischen Validierung mit einem Wert aus dieser Menge übereinstimmen, sonst wird dies als Benutzerfehler gewertet. Die Wertemenge der Domain wird für die Ausgabe an das Stylesheet geschickt. So können z.B. in formularbasierten Anwendungen Listboxen realisiert werden.

List

Eine List hat folgende zusätzliche Eigenschaften:

- **Minimum Size**: Für die Strukturvalidierung, z.B. bei SOAP-Anfragen, kann eine Mindestgröße einer List festgelegt werden. Ein Unterschreiten wird als Anwendungsfehler gewertet.

- **Default Size**: Eine List wird mit allen ihren Listenelementen ausgegeben, mindestens jedoch mit so vielen, wie hier als *default size* definiert wurde. Hat eine List in der Session also keine (z.B. bei initialer Ausgabe) oder weniger Listenelemente, als in *default size* angegeben, werden der Ausgabe automatisch leere Listenelemente hinzugefügt. Somit läßt sich z.B. bei einem tabellenartigen Formular die Anzahl der Tabellenzeilen für die Anzeige festlegen.

- **Maximum Size**: Für die Strukturvalidierung, z.B. bei SOAP-Anfragen, kann eine Maximalgröße einer List festgelegt werden. Ein Überschreiten wird als Anwendungsfehler gewertet.

5.5.2 Datatype View

Atome haben Dateninhalte, die bestimmten frei definierbaren Datentypen genügen müssen. Diese Objekt-Dateninhalte sind sprachunabhängig. Dateninhalte von Atomen werden jedoch bei der Ein- und Ausgabe von bzw. in sprachspezifische Stringrepräsentationen umgewandelt.

Datentypen werden auf beliebige Java-Klassen abgebildet. Sie müssen jedoch eindeutige Stringrepräsentationen besitzen, wenn sie zur Ein- und Ausgabe verwendet werden.

Datentypen werden in der *Datatype View* definiert. Sie werden über die rechte Maustaste mit dem entsprechenden Menüeintrag angelegt und gelöscht. Sie können in einem Atom oder als Konstante genutzt werden. Sobald ein Datentyp verwendet wird, erhält er ein Schloßsymbol und kann nicht mehr gelöscht werden.

Unter Umständen können Datentypen mit Fehlersymbolen gekennzeichnet sein. Die genauen Beschreibungen sind dann in der *Task View* zu finden bzw. erscheinen als Hinweistext in der Statuszeile. Fehler müssen unbedingt gelöst werden, bevor die *zen*-Anwendung gestartet wird.

- **Name**: Frei definierbare Bezeichnung zur Anzeige in der *Data Model View*
- **Target Class**: Java-Klasse, die den Datentyp repräsentiert
- **Converter**: Java-Klasse, die die Konvertierung von sprachspezifischen Stringrepräsentationen in sprachunabhängige Objekte bzw. die Konvertierung von Objekten in sprachspezifische Stringrepräsentationen implementiert. Eine solche Klasse muß das Interface *AtomParsing* implementieren.
- **Error Message**: Fehlermeldung (Data Error), die als Benutzerfehler verwendet wird, wenn eine Stringrepräsentation nicht in ein Objekt des entsprechenden Datentyps konvertiert werden kann. Wird z.B. bei falsch formatierten Eingaben von Benutzern in ein Formular-Feld angezeigt.

5.5.3 Domain View

In der *Domain View* werden Wertemengen definiert, die Atomen zugeordnet werden können. Eine Domäne erfüllt dabei zwei Funktionen: Während der syntaktischen Validierung wird sichergestellt, daß Atome mit zugewiesenen Domänen nur Eingaben erhalten, die in der Wertemenge definiert sind. Bei der Ausgabe wird die Wertemenge mit an das XSL-Stylesheet gesendet, das Atom referenziert dort den ausgewählten Wert. So lassen sich z.B. bei browserbasierten Web-Anwendungen Listboxen oder Radio-Button-Gruppen realisieren.

Domänen bestehen grundsätzlich aus stringbasierten Key/Value-Paaren. Der Key ist ein sprachunabhängiger String, der dem Datentyp des Atoms entsprechen muß (in der Regel also *String* oder andere Datentypen mit sprachunabhängiger Stringrepräsentation, z.B. *Integer* oder entsprechend implementierte eigene Datentypen), der Value ist der sprachspezifische Wert seines Keys. Für jeden Key wird damit jeweils ein Wert für jede unterstützte Sprach-Länderkombination definiert.

Es gibt zwei Arten von Domänen: repository- und entwicklerdefinierte Domänen. Bei ersteren werden alle Key/Value-Paare im Repository definiert, bei zweiteren übernimmt eine Java-Klasse die Definition der Key/Value-Paare. Die Java-Klasse kann dann z.B. die Key/Value-Paare aus einer Datenbank lesen. Einer entwicklerdefinierten Domäne kann jedoch ein zusätzliches Key/Value-Paar vorangestellt werden, das im Repository definiert wird (z.B. ein leerer Key und als Value eine Auswahlaufforderung, z.B. „bitte wählen“). Einer entwicklerdefinierten Domäne können auch Eingabeparameter zugeordnet werden. Diese müssen in Anzahl und Klassen mit den entsprechenden Parametern des Konstruktors ihrer Java-Klasse korrespondieren.

Domänen werden über die rechte Maustaste über den entsprechenden Menüeintrag angelegt und gelöscht. Sobald eine Domäne verwendet wird, erhält sie ein Schloßsymbol und kann nicht mehr gelöscht werden.

Unter Umständen können Domänen mit Fehlersymbolen gekennzeichnet sein. Die genauen Beschreibungen sind dann in der *Task View* zu finden bzw. erscheinen als Hinweistext in der Statuszeile. Fehler müssen unbedingt gelöst werden, bevor die *zen*-Anwendung gestartet wird.

Reiter Properties

Hier können grundsätzliche Eigenschaften von Domänen definiert werden.

- **Global**: Eine so markierte Domäne kann einmal erstellt und in allen Anwendungen verwendet werden, die im gleichen Repository liegen. Domänen können nur als *global* definiert werden, wenn sie keinen Bezug zu einer bestimmten Anwendung haben, also z.B. nicht über Element-Argumente mit dem Datenmodell einer bestimmten Anwendung verbunden sind. Wird eine als *global* definierte Domäne bereits verwendet, kann die *global*-Eigenschaft nicht zurückgenommen werden.
- **Error Message**: Fehlermeldung (Data Error), die als Benutzerfehler verwendet wird, wenn ein Atom bei der syntaktischen Validierung einen Wert enthält, der nicht in der Wertemenge der ihm zugewiesenen Domäne definiert ist.
- **Custom Class**: Erlaubt die Auswahl einer eigenen Java-Klasse für entwicklerdefinierte Domänen. Die Java-Klasse muß das Interface *Domain* implementieren. Wird einer repositorydefinierten Domäne eine *Custom Class* zugewiesen, werden alle bisher zugeordneten Key/Value-Paare gelöscht. Wird die *Custom Class* einer entwicklerdefinierten Domäne gelöscht, werden alle zugeordneten Argumente mitentfernt.

- **Lifecycle**: Beschreibt den Instantiierungszeitpunkt bzw. den Lebenszyklus einer entwicklerdefinierten Domäne:
  - *system*: Die Domäne wird bei der Initialisierung des Repository instantiiert und hängt an der Lebensdauer des Repository-Caches. Wird dieser gelöscht und neu geladen, wird auch die Domäne neu instantiiert. Eine Domäne mit *system lifecycle* kann nur mit Konstanten als Argumenten versorgt werden.
  - *session*: Die Domäne wird bei der ersten Benutzung innerhalb einer Benutzer-Session instantiiert und hängt an der Lebensdauer der Session. Läuft diese auf einen Timeout, wird die Domäne gelöscht. Eine Domäne mit *session lifecycle* kann mit allen Argumenten versorgt werden (wenn sie nicht als *global* definiert wurde). Es muß jedoch bei zugewiesenen Element-Argumenten sichergestellt sein, daß diese bereits bei der Instantiierung einer so definierten Domäne zur Verfügung stehen.
  - *state*: Eine so definierte Domäne wird instantiiert, bevor sie in einem bestimmten State Node ausgegeben wird. Dieselbe Instanz wird zur syntaktischen Validierung des darauffolgenden Requests benutzt. Sie bietet also Konsistenz zwischen Ausgabe und nachfolgender Validierung. Eine Domäne mit *state lifecycle* kann mit allen Argumenten versorgt werden (wenn sie nicht als *global* definiert wurde). Es muß jedoch bei zugewiesenen Element-Argumenten sichergestellt sein, daß diese bereits bei der Instantiierung einer so definierten Domäne zur Verfügung stehen.
  - *transition*: Eine so definierte Domäne wird instantiiert, wenn sie zur syntaktischen Validierung eines Requests benötigt wird. Dieselbe Instanz wird dann zur Ausgabe benutzt. Sie bietet also Konsistenz zwischen Validierung und nachfolgender Ausgabe. Wenn z.B. im Umkehrschluß in einer browserbasierten Web-Anwendung dem Benutzer eine *transition*-Domäne angezeigt wird, dieser einen Wert auswählt und einen neuen Request erzeugt, wird sie während der Validierung neu instantiiert und kann ihre Wertemenge so verändern, daß der gewählte Wert nicht mehr in der Domäne vorkommt. Ein Benutzerfehler ist die Folge. Eine Domäne mit *transition lifecycle* kann mit allen Argumenten versorgt werden (wenn sie nicht als *global* definiert wurde). Es muß jedoch bei zugewiesenen Element-Argumenten sichergestellt sein, daß diese bereits bei der Instantiierung einer so definierten Domäne zur Verfügung stehen
- **Constructors**: Anzeigefeld für alle verfügbaren Konstruktoren einer entwickler-definierten Domäne. Dient als Orientierung bei der Argument-Versorgung. Derzeit nicht verwendete bzw. passende Konstruktoren werden mit einem roten Kreuz markiert. Bei Auswahl einer passenden Argument-Kombination verschwindet das rote Kreuz des entsprechenden Konstruktors.
- **Arguments**: Der Domäne lassen sich hier Argumente für den Konstruktor zuordnen. Je nachdem, ob die Domäne als *global* definiert ist, bzw. abhängig vom *lifecycle* stehen bis zu drei Varianten von Argumenten zur Verfügung. Diese können mit dem Pfeil nach rechts der Domäne zugewiesen bzw. mit dem Pfeil nach links wieder entfernt werden. Mit den Pfeilen nach oben/unten kann die Reihenfolge der Argumente verändert werden.
  - *Arguments (Element-Argumente)*: Der Domäne lassen sich Datenelemente als Argumente zuweisen. Es sind alle Elemente erlaubt bis auf Elemente, die sich unterhalb einer List befinden, da eine List zur Laufzeit mehrere Kinder haben kann. Wird ein Element als Argument ausgewählt, kann in der Spalte *Call By* bei Atomen noch der Übergabemodus ausgewählt werden. Compositions und Lists werden immer als *reference* übergeben.
    - *value*: Objekt-Dateninhalt eines Atoms (sprachunabhängig). Paßt auf Konstruktor-Parameter, wenn dessen Java-Klasse mit dem Datentyp des Atoms übereinstimmt. Im Normalfall ausreichend.
    - *reference*: das FOM-Element selbst. Paßt je nach Elementtyp auf Konstruktor-Parameter der Klassen *Element, Atom, Composition* bzw. *List* aus der FOM API. Dann zu verwenden, wenn manuelle Navigation über die Sessionelemente, z.B. über Listenelemente nötig ist oder strukturelle Änderungen an dem Sessiondaten-Baum durchgeführt werden sollen.
    - *value rep*: sprachspezifische Stringrepräsentation des Dateninhalts eines Atoms. Paßt auf Konstruktor-Parameter der Klasse *String*. Nur in Ausnahmefällen zu verwenden, wenn z.B. die Stringrepräsentationen in die Domänen-Values integriert werden müssen.
  - *Constant (Konstanten-Argumente)*: Einer Domäne können Konstanten als Argumente zugeordnet werden. Konstanten können in dem Auswahlfenster über die rechte Maustaste angelegt und gelöscht werden. Benutzte Konstanten werden mit einem Schloßsymbol gekennzeichnet und können nicht gelöscht werden. Konstantendefinitionen sind global, d.h. sie stehen allen Anwendungen innerhalb eines Repository zur Verfügung. Eine Konstante wird mit einer beliebigen Bezeichnung versehen, bekommt einen Datentyp zugeordnet und einen stringbasierten Wert im Format der Default-Locale. Ein Konstanten-Argument paßt auf einen Konstruktor-Parameter, wenn dieser mit der Java-Klasse des Datentyps der Konstante übereinstimmt.
  - *Field Function (Feldfunktions-Argumente)*: Einer Domäne können Feldfunktionen als Argument zugeordnet werden. Hier handelt es sich um fest definierte Funktionen, die von der *zen Engine* zur Verfügung gestellt werden. Manche Feldfunktionen können ihrerseits mit einem Argument versehen werden. Dies

kann in der Spalte *FF-Argument* angegeben werden. Feldfunktionen passen auf Konstruktor-Parameter, wenn deren Java-Klasse mit der Rückgabe-Klasse der Feldfunktion übereinstimmt.

Reiter Contents

Hier werden bei repositorydefinierten Domänen alle Key/Value-Paare erstellt, bei entwicklerdefinierten Domänen kann ein einziges Key/Value-Paar definiert werden, das der Domäne vorangestellt wird.

In dem Fenster *Key* können mit der rechten Maustaste über den entsprechenden Menüeintrag neue Keys angelegt werden. Ein Key muß dem Datentyp des Atoms entsprechen, dem die Domäne später zugeordnet wird. Im Fenster *Locale* werden auf die gleiche Weise Locales angelegt. Wird eine Locale selektiert, kann im Fenster *Key/Value* jedem Key in der selektierten Locale ein Value zugeordnet werden. Der Value ist in der Regel die locale-spezifische Beschreibung des Keys, die zur Darstellung genutzt wird. Mit dem Pfeil nach oben bzw. unten können die Wertemengen in der selektierten Locale sortiert werden. Jede Locale kann eine eigene Sortierung haben. Wird ein Key gelöscht, wird der Key aus jeder Locale gelöscht. Wird eine Locale gelöscht, werden alle Key/Value-Paare der Locale gelöscht.

5.6 Erstellen der Geschäftslogik

Die Geschäftslogik einer *zen*-Anwendung wird in Java-Klassen implementiert. Diese Klassen stellen Methoden als Einsprungpunkte zur Verfügung, die in der *Business Logic View* als Operation mit dem Workflowmodell und/oder Datenmodell der *zen*-Anwendung verknüpft werden. Als Einsprungpunkt kann jede *public* Methode verwendet werden, normalerweise sollte die Methode jedoch zusätzlich als *static* definiert werden. Anderenfalls wird die Klasse bei jedem Aufruf eines ihrer Einsprungpunkte neu instantiiert.

Einer Operation werden Eingabe- und Rückgabeparameter zugeordnet. Diese müssen in Anzahl und Klassen mit den entsprechenden Parametern der Java-Methode korrespondieren. Besitzt eine Operation mehr als einen Rückgabeparameter, wird dies in der Java-Methode als Array abgebildet. Argumente können Elemente aus dem Datenmodell, Konstanten oder von der *zen Engine* fest definierte sog. Feldfunktionen sein. Wird eine Operation aufgerufen, werden die zugeordneten Argumente automatisch aus den Sessiondaten ermittelt. Rückgabewerte einer Operation werden ebenfalls automatisch in die Sessiondaten eingefügt.

Besondere Erwähnung verdient die Behandlung von Operationen mit Parametern, die mit Elementen unterhalb von Listen verbunden sind. Die Methode einer solchen Operation wird so oft aufgerufen, wie es Elemente unterhalb der Liste in der Session gibt. Die Parameter werden für jeden Aufruf aus dem gerade bearbeiteten Element der Liste ermittelt. Dabei können sich einige Eingabe- oder Rückgabeparameter unterhalb einer Liste, andere oberhalb einer Liste befinden. Die Parameter unterhalb der Liste werden dann für jeden Aufruf mit den Inhalten des gerade bearbeiteten Elements der Liste gefüllt, die Parameter oberhalb der Liste ändern sich nicht. Nicht erlaubt ist die Zuordnung von Parametern einer Operation zu benachbarten Listen.

In unserem Beispiel-Datenmodell definieren wir auf einer Java-Methode
String ermittleWertpapierName(Integer wkn)
eine Operation und verknüpfen sie mit dem Element /data/lfd-orders/lfd-order/wkn *als Eingabe-Argument und* /data/lfd-orders/lfd-order/wertpapier-name *als Rückgabe-Argument. Liegen zum Zeitpunkt des Aufrufes dieser Operation in der Session z.B. zehn laufende Orders vor, d.h. die Liste* lfd-orders *hat zehn Kindelemente* lfd-order*, wird die Java-Methode der Operation zehnmal aufgerufen. Die Ergebnisse werden jeweils in dasjenige Element* wertpapier-name *geschrieben, das Kind der gleichen Composition* lfd-order *ist, wie das Eingabe-Argument* wkn.

Operationen können Benutzerfehler erzeugen, indem sie eine *OperationException* werfen (Validation Rules werfen stattdessen eine *ValidationRuleException*). Diese Exceptions können mit einem Bezeichner versehen werden, der zur Auswahl einer Fehlermeldung dient. Wirft eine Operation einen Laufzeitfehler, wird dies als Anwendungsfehler gewertet.

Anwendung der Service-API

Bei der Implementierung der Geschäftslogik werden üblicherweise verschiedene technische Services wie beispielsweise JDBC oder Logging benötigt. Diese werden von der Service-API der *zen Platform* zur Verfügung gestellt. Jede Implementierung einer Operation kann ohne Einschränkung und unabhängig vom Deployment über die Service-API auf die vorhandenen Dienste zugreifen.

Die einzelnen Services und deren Anwendung sind ausführlich in der *zen Engine Referenz* dokumentiert.

Business Logic View

Die *Business Logic View* erlaubt die Definition von Operationen. Im linken Teil der View sind alle definierten Operationen dargestellt. Mittels Karteireitern läßt sich zwischen einer Übersicht über alle Operationen, zwischen Computation Rules, Validation Rules, Decision Operations und Workflow Operations umschalten. Die Übersicht ist als Tabelle dargestellt, in der alle Operationen mit ihrer Beschreibung, Klasse, Methode und Operation-Typ aufgelistet werden. In den anderen Karteireitern werden sie wie im *Package Explorer* als Baumdarstellung angezeigt. Unterhalb der Java-Methoden in der Package-Hierarchie sind die dazu definierten Operationen aufgeführt.

Operationen werden über die rechte Maustaste über den entsprechenden Menüeintrag angelegt und gelöscht. Beim Anlegen wird ein Dialog geöffnet, in dem eine Beschreibung der Operation, ihre Java-Klasse und Methode, der Operationstyp und weitere Eigenschaften angegeben werden. Zu einer bereits angelegten Operation können ebenfalls über die rechte Maustaste (*Add*) zusätzliche Operationen hinzugefügt werden, um diese z.B. mit anderen Argumenten versehen zu können. Die Operationsbeschreibung kann in jeder Kartei geändert werden, der Operationstyp nur in der Übersichtskartei. Decision Operations können nur auf

Decision Nodes verwendet werden, Workflow Operations in Actions, Transitions und auf State Nodes. Sobald eine Decision oder Workflow Operation verwendet wird, erhält sie ein Schloßsymbol und kann nicht mehr gelöscht werden.

In der Baumdarstellung kann mit der rechten Maustaste direkt in die entsprechende Java-Klasse bzw. Java-Methode gesprungen werden.

Properties

- **Stop On Error**: Tritt bei Ausführung der Operation ein Benutzerfehler auf, werden normalerweise trotzdem alle weiteren Operationen gleichen Typs ausgeführt, um Benutzerfehler zu sammeln (z.B. bei einem Benutzerfehler in einer Validation Rule alle weiteren Validation Rules, bei einer Action Operation alle nachfolgenden Action Operations, bei einer Transition Operation alle nachfolgenden Transition Operations usw.). Wird eine Operation als *stop on error* definiert, wird die weitere Ausführung von gleichartigen Operationen sofort unterbunden. Insbesondere wird auch bei einer *erroraware*-Action die Ausführung von weiteren Operationen abgebrochen.

- **Call With <null>**: Eine Operation, die als *call with <null>* definiert ist, wird auch aufgerufen, wenn einer der Parameter *<null>* ist. Ein Parameter ist immer dann *<null>*, wenn ein als Argument benutztes Daten-Element nicht in der Session vorhanden ist oder wenn ein Atoms, das per *call by value* übergeben wird, keinen Dateninhalt hat. Die Java-Methode muß dann entsprechend berücksichtigen, daß ein oder mehrere Parameter *null* sein können.

  Workflow Operations sind beim Anlegen standardmäßig immer *call with <null>*, da Workflow Operations normalerweise zu einem festgelegten Zeitpunkt ausgeführt werden sollen. Entfernt man das Flag in diesem Fall, bekommt eine Workflow Operation eine Optionalität ähnlich einer Business Rule.

  Business Rules sind werden standardmäßig ohne *call with <null>* angelegt, da sie explizit von den Daten gesteuert werden und bei fehlenden Daten normalerweise keine Ausführung stattfinden sollte.

- **Aggregation**: Eine Operation, die als Eingabeparameter Elemente unterhalb einer Liste hat, wird so oft aufgerufen, wie es Elemente der Liste in der Session gibt und produziert normalerweise entsprechend viele Rückgabewerte. Wird eine Operation als *Aggregation* definiert, muß die Java-Klasse der Operation das Interface *AggregationOperation* implementieren und die Operation *void* zurückgeben. Die Java-Klasse wird dann vor Ausführung instantiiert und für jedes Listen-Element wird die definierte Methode aufgerufen. In der Instanz der Operationsklasse können Ergebnisse aggregiert werden. Nachdem alle die Operation für alle Listen-Elemente aufgerufen wurde, wird die Methode *getResult* aufgerufen und deren Rückgabe in die Sessiondaten zurückgeschrieben. Das modellierte Rückgabe-Argument der Aggregation liegt in der Regel oberhalb der Liste.

- **Priority**: Nur für Business Rules (Computation und Validation Rules). Die Ausführbarkeit bzw. Ausführungsreihenfolge von Business Rules kann im wesentlichen erst zur Laufzeit ermittelt werden. Um innerhalb der aufzurufenden Business Rules die Reihenfolge dennoch in einem gewissen Rahmen zu steuern, kann eine Priorität angegeben werden. Je niedriger der angegebene Wert, desto höher die Priorität.

  Bei Validation Rules legt eine Priorisierung explizit die Reihenfolge fest.

  Bei Computation Rules wird die Reihenfolge jedoch zuerst durch die Abhängigkeit untereinander, die anhand deren Eingabe- und Rückgabeparameter ermittelt wird, bestimmt. Darüber hinaus wird bei unabhängigen oder bezüglich ihrer Abhängigkeiten gleichwertigen Computation Rules diejenige mit der höchsten Priorität (mit dem niedrigsten Wert) zuerst ausgeführt.

Error Message

Um Fehlerzustände zu kommunizieren, kann jede Validation Rule eine *ValidationRuleException*, jede andere Operation eine *OperationException* werfen. Eine Operation erzeugt so einen Benutzerfehler. Im Unterfenster „Available Resources“ werden alle in der *Resource View* definierten *Operation Errors* angezeigt. Diese können mit dem Pfeil nach rechts einer Operation zugewiesen bzw. mit dem Pfeil nach links wieder entfernt werden. Wenn nur eine Fehlerkondition für eine Operation existiert, wird eine Fehlermeldung mit dem Standard-Namen *error* erwartet. Existieren mehrere unterschiedliche Fehlerkonditionen innerhalb einer Operation, kann im Java-Code jeweils über den Exception-Konstruktor ein Name für die zu verwendende Fehlermeldung festgelegt werden. Die zugeordneten Fehlermeldungen müssen dann passend bezeichnet sein, anstatt den Standard-Namen *error* zu verwenden.

Arguments/Return Values

Einer Operation lassen sich hier Eingabe- und Rückgabe-Argumente für ihre Java-Methode zuordnen. Hierzu stehen drei Varianten von Argumenten zur Verfügung. Im rechten oberen Teil innerhalb der Gruppe *Arguments/Return Values* werden Eingabe-Argumente definiert, in der unteren Hälfte Rückgabe-Argumente. Diese können der Operation jeweils mit dem Pfeil nach rechts zugewiesen bzw. mit dem Pfeil nach links wieder entfernt werden. Mit den Pfeilen nach oben/unten kann die Reihenfolge der jeweiligen Argumente verändert werden.

- *Arguments/Return Values (Daten-Elemente)*: Der Operation werden Elemente aus dem Datenmodell als Argumente zugewiesen. Bei Atomen kann in der Spalte *Call By* noch der Übergabemodus ausgewählt werden. Compositions und Lists werden immer als *reference* übergeben.

    - *value*: Objekt-Dateninhalt eines Atoms (sprachunabhängig). Paßt auf Methoden-Parameter, wenn dessen Java-Klasse mit dem Datentyp des Atoms übereinstimmt. Im Normalfall ausreichend.
    - *reference*: das FOM-Element selbst. Paßt je nach Elementtyp auf Methoden-Parameter der Klassen *Element, Atom, Composition* bzw. *List* aus der FOM API. Dann zu verwenden, wenn manuelle Navigation über die Sessionelemente, z.B. über Listenelemente nötig ist oder strukturelle Änderungen an dem Sessiondaten-Baum durchgeführt werden sollen.
    - *value rep*: sprachspezifische Stringrepräsentation des Dateninhalts eines Atoms. Paßt auf Methoden-Parameter der Klasse *String*. Nur in Ausnahmefällen zu verwenden, da Operationen in der Regel sprachunabhängig implementiert werden.
- *Constant (Konstanten)*: Einer Operation können Konstanten als Argumente zugeordnet werden. Konstanten können im Auswahlfenster mit der rechten Maustaste angelegt und gelöscht werden. Benutzte Konstanten werden mit einem Schloßsymbol gekennzeichnet und können nicht mehr gelöscht werden. Konstanten-Definitionen sind global, d.h. sie stehen allen Anwendungen innerhalb eines Repository zur Verfügung. Eine Konstante wird mit einer beliebigen Bezeichnung versehen, bekommt einen Datentyp zugeordnet und einen stringbasierten Wert im Format der Default-Locale. Ein Konstanten-Argument paßt auf einen Methoden-Parameter, wenn dieser mit der Java-Klasse des Datentyps der Konstante übereinstimmt.
- *Field Function (Feldfunktionen)*: Einer Operation können Feldfunktionen als Argument zugeordnet werden. Hier handelt es sich um fest definierte Funktionen, die von der *zen Engine* zur Verfügung gestellt werden. Manche Feldfunktionen können ihrerseits mit einem Argument versehen werden. Dies kann in der Spalte *FF-Argument* angegeben werden. Feldfunktionen passen auf Methoden-Parameter, wenn deren Java-Klasse mit der Rückgabe-Klasse der Feldfunktion übereinstimmt.

5.7 Erstellen von Ressourcen

Sprachspezifische String-Ressourcen, z.B. Texte oder URLs, können Elementen, State Nodes, Actions und Operations zur Vertextung zugeordnet werden. Element-Resources können z.B. dazu dienen, Formular-Elemente mit Bezeichnern, Beschriftungen oder erklärenden Texten zu versehen. State-Resources werden für die Vertextung ganzer State Nodes verwendet, z.B. um Formular-Seiten mit Überschriften und Textblöcken auszustatten. Action-Resources können z.B. dazu dienen, Button-Beschriftungen (als Text oder Bild-URL) zu erzeugen. Als Operation Error bzw. Data Error werden Fehlermeldungen definiert, die dem Endnutzer angezeigt werden können. Operation Errors sind Fehlermeldungen für Operationen, die eine *OperationException* bzw. eine *ValidationRuleException* werfen. Data Errors sind Fehlermeldungen bei Eingabefehlern, die Datatypes, Domänen und Atomen zugeordnet werden (z.B. zur Anzeige eines Fehlers bei verletzter *mandatory*-Eigenschaft eines Atoms).

Ressourcen können zusätzlich mit dem Datenmodell verknüpft werden, um Dateninhalte dynamisch in Beschriftungen einzubetten. Einer Ressource können hierzu Argumente zugeordnet werden. Ressourcen, die Argumente entgegennehmen können, müssen mit Platzhaltern für die Argumente formuliert werden. Die Argumente werden dann zur Laufzeit in die Platzhalter eingefügt. Die Syntax orientiert sich an der Klasse *java.text.MessageFormat* und kann in der Java-Dokumentation von Sun nachgelesen werden.

Einige Beispiele von klassischen MessageFormat-*Möglichkeiten*:

`„Die Order Nr. {0} wurde am {1, date, full} um {1, time} entgegengenommen.“`

`„Aktuell {0,choice,1#liegt eine Order|1< liegen {1} Orders} vor.“`

Wenn der Format-Typ *choice* verwendet wird, werden außer den in der Klasse *MessageFormat* definierten *double*-Intervallgrenzen noch folgende zusätzliche Bezeichner als Intervallgrenzen erlaubt, die auf *double*-Werte abgebildet werden:

- `null# ≡ -∞#, null< ≡ -DOUBLE.MAX_VALUE#`
- `false ≡ 0, true ≡ 1`
- `"string" ≡ x` , wobei *string* eine beliebige Zeichenkette sein kann und *x* die 0-basierte Reihenfolge des Auftretens pro Ressource und Argument-Platzhalter ist.

Mit den Intervallbezeichnern null# *und* null< *läßt sich* null *bzw.* != null *ausdrücken*:

`„Das Element {0,choice,null#fehlt|null<ist vorhanden}.“`

Argumente, die null *sind, werden auf -∞ abgebildet und treffen somit auf die Intervallgrenze* null# *zu. Argumente, die* != null *sind, werden auf* double *abgebildet. Dies geschieht entweder durch ihren natürlichen* double-*Wert, wenn sie von* Number *abgeleitet sind, oder durch eine oben definierte Abbildung. Ansonsten nehmen sie den Wert* -DOUBLE.MAX_VALUE *an. Argumente, die* != null *sind, passen somit immer mindestens auf die Intervallgrenze* null<.

Intervallbezeichner lassen sich auch kombinieren. Dabei muß beachtet werden, daß die Intervallgrenzen in aufsteigender Reihenfolge vorliegen müssen. So sind z.B. folgende Kombinationen möglich:

`„Der Zustand ist {0,choice,null#unbekannt|false#ungültig|true#gültig}.“`

oder:

`„Die Zahl ist {0,choice,null#unbekannt|null<negativ|0#0|0<positiv}.“`

Wenn in einer Ressource mit dem Format-Typ *choice* stringbasierte Intervallgrenzen verwendet werden, können also String-Argumente zugeordnet werden. Zur Laufzeit wird dann die entsprechende Alternative bei Übereinstimmung der String-Argumente ausgewählt. Bei der Erstellung einer solchen Ressource ist zu beachten, daß die Reihenfolge der stringbasierten Intervallgrenzen pro Ressource und Argument immer gleich sein muß.

Eine korrekt formatierte parametrisierte Ressource ist z.B.:

„`Die Order wurde {0,choice,"accept"#akzeptiert|"reject"#zurückgewiesen}.`“

und:

„`{0,choice,"a"#A|"b"#B|"c"#C}{1,choice,"b"#X|"a"#Y}`“

und:

„`{0,choice,"a"#A|"b"#B|"c"#C}{0,choice,"a"#X|"c"#Y}`“

aber nicht:

„`{0,choice,"a"#A|"b"#B|"c"#C}{0,choice,"b"#X|"a"#Y}`“

da innerhalb derselben Ressource für einen Argumentplatzhalter (`0`) *unterschiedliche Reihenfolgen der Intervallbegrenzer verwendet werden*

Resource View

Die *Resource View* erlaubt die Definition von Ressourcen. Im linken Teil der View sind alle definierten Ressourcen mit den Inhalten der aktuellen Entwicklungslocale dargestellt. Diese kann in der Toolbar am oberen Rand des *zen Developer* geändert werden. Mittels Karteireitern läßt sich zwischen einer Übersicht über alle Ressourcen, zwischen Element-, State- und Action-Resources sowie zwischen Operation- und Data Errors umschalten. Ressourcen werden mit der rechten Maustaste über den entsprechenden Menüeintrag angelegt und gelöscht. Die Namen der Ressourcen können hier genauso editiert werden wie die Inhalte der Ressourcen in der Entwicklungslocale. Wird eine Ressource verwendet, wird sie mit einem Schloßsymbol versehen und kann nicht mehr gelöscht werden.

Unter Umständen können Ressourcen bzw. Ressourcen-Inhalte mit Fehlersymbolen gekennzeichnet sein. Die genauen Beschreibungen sind in der *Task View* zu finden bzw. erscheinen als Hinweistext in der Statuszeile. Fehler müssen unbedingt gelöst werden, bevor die *zen*-Anwendung gestartet wird.

Properties

- **Global**: Eine so markierte Ressource kann einmal erstellt und in allen Anwendungen verwendet werden, die im gleichen Repository liegen. Ressourcen können nur als *global* definiert werden, wenn sie keinen Bezug zu einer bestimmten Anwendung haben. Sie dürfen also z.B. nicht über Element-Argumente mit dem Datenmodell einer bestimmten Anwendung verbunden sein. Wird eine als *global* definierte Ressource bereits verwendet, kann die *global*-Eigenschaft nicht zurückgenommen werden.
- **Type**: erlaubt das Einordnen der Ressource in eine andere Kategorie

Values

Hier können Inhalte von Ressourcen definiert werden. Jeder Ressource können beliebig viele Locale/Inhalt-Paare zugeordnet werden. Über die rechte Maustaste werden über den entsprechenden Menüeintrag neue Locale/Value-Paare angelegt bzw. wieder gelöscht. Es ist darauf zu achten, daß in einer Anwendung alle Ressourcen für alle vorkommenden Locales angelegt sein müssen.

- **Arguments**: Einer Ressource, die auf *MessageFormat* basiert, lassen sich hier Argumente zuordnen. Hierzu stehen drei Varianten von Argumenten zur Verfügung. Diese können mit dem Pfeil nach rechts der Ressource zugewiesen bzw. mit dem Pfeil nach links wieder entfernt werden. Mit den Pfeilen nach oben/unten kann die Reihenfolge der Argumente verändert werden.
  - *Arguments (Element-Argumente)*: Der Ressource lassen sich Atome als Argumente zuweisen, sofern sie sich nicht unterhalb einer List befinden, da eine List zur Laufzeit mehrere Kinder haben kann. In der Spalte *Call By* kann noch der Übergabemodus ausgewählt werden.
    - *value*: Objekt-Dateninhalt eines Atoms (sprachunabhängig), geeignet für den Format-Typ *choice* und andere Formate (z.B. *number*, *date*, *time*). Das ggf. in den Formaten angegebene jeweilige Subformat bestimmt die genaue Formatierung des Inhalts.
    - *reference*: das FOM-Element selbst. Dieser Übergabemodus ist normalerweise nur dann sinnvoll, wenn in einer Ressource der Formattyp *choice* verwendet wird und das Argument ausschließlich auf die Intervallgrenzen *null#* bzw. *null*< angewendet wird.
    - *value rep*: sprachspezifische Stringrepräsentation des Dateninhalts eines Atoms. Hier kann direkt die vom Datentyp-Konvertierer erzeugte Stringrepräsentation in den Ressourcen-Inhalt integriert werden.
  - *Constant (Konstanten-Argumente):* Einer Ressource können Konstanten als Argumente zugeordnet werden. Konstanten können in dem Auswahlfenster mit der rechten Maustaste angelegt und gelöscht werden. Benutzte Konstanten werden mit einem Schloßsymbol gekennzeichnet und können nicht mehr gelöscht werden. Konstanten-Definitionen sind global, d.h. sie stehen allen Anwendungen innerhalb eines Repository zur Verfügung. Eine Konstante wird mit einer beliebigen Bezeichnung versehen, bekommt einen Datentyp zugeordnet und einen stringbasierten Wert im Format der Default-Locale. Das Konstanten-Argument wird zur Laufzeit über die Java-Klasse des Datentyps der Konstante in die Ressource eingefügt.

- *Field Function (Feldfunktions-Argumente):* Einer Ressource können Feldfunktionen als Argument zugeordnet werden. Hier handelt es sich um fest definierte Funktionen, die von der *zen Engine* zur Verfügung gestellt werden. Manche Feldfunktionen können ihrerseits mit einem Argument versehen werden. Dies kann in der Spalte *FF-Argument* angegeben werden. Feldfunktionen werden mit der Rückgabe-Klasse der Feldfunktion in die Ressource eingefügt.

5.8 Debugging einer Anwendung

Um die *zen Engine* und damit eine *zen*-Anwendung im Debugger zu starten, legt man einmalig einen Debug-Launch an. Über das Menü *Run->Debug...* wird der Konfigurationsdialog aufgerufen. In der Kategorie *Java Application* wird mit *New* eine neue Konfiguration angelegt. Als Projekt wird das Java-Projekt ausgewählt, in dem die *zen*-Anwendung angelegt wurde. Die Checkbox *Include external jars when searching for a main class* wird aktiviert. Mit Klick auf *Search...* wird der Suchdialog für die Startklasse angezeigt. Dort wird die Klasse *TomcatStart* aus dem Archiv *zenapi.jar* ausgewählt. Jetzt kann der Debug-Vorgang gestartet werden.
Sofern Eclipse die Klasse *TomcatStart* nicht findet, trägt man manuell *de.zeos.zen.tomcat.TomcatStart* ein.

Der Debug-Launch wird über den Button *Debug* gestartet. Sofern die Anwendung noch nicht völlig frei von Fehlern oder Warnungen ist, erscheint danach eine Dialogbox mit einer entsprechenden Warnmeldung.

Nach dem Start können Requests an die *zen*-Anwendung gesendet und im Debugger untersucht werden. Für browser-basierende Frontends bietet der *zen Editor* auf *entry*- oder *defaultentry*-State Nodes ein Kontextmenü an. Mit der rechten Maustaste und Auswahl des entsprechenden Menüpunktes wird der in Eclipse konfigurierte Browser mit der richtigen URL gestartet. Je nach Betriebssystem bringt sich Eclipse nach dem Start des Browsers eventuell selbständig wieder in den Vordergrund und überdeckt dadurch den soeben gestarteten Browser.

Im *zen Editor* können auf allen Workflow-Elementen Breakpunkte für den Debug-Vorgang gesetzt werden. Dazu muß in der Toolbar oder Menü *Zen* die Schaltfläche *Mouseover Introspection* aktiviert werden. Für verschiedene Workflow-Elemente stehen verschiedene Breakpunkt-Varianten zur Verfügung. Die graphischen Breakpunkte werden mit einem Kreis-Symbol gekennzeichnet, in denen ein Buchstabenkürzel steht. Aktivierte Breakpunkte werden blau, deaktivierte grau dargestellt. Die *zen Engine* bleibt in den jeweils aktivierten Breakpunkten stehen. Die einzelnen Varianten stehen für folgende Bearbeitungsschritte:

Ausgehende Transition eines State Nodes

- **AC**: Action Operations
- **TR**: Transition Operations

Ausgehende Transition eines Decision Nodes

- **PD**: Post Decision Operations

State Node

- **PR**: Pre State Operations
- **RV**: Request Validation
- **CR**: Computation Rules
- **VR**: Validation Rules
- **PO**: Post State Operations

Decision Node

- **DE**: Decision Operations

Beim Abarbeiten des Workflows werden alle aktuell durchlaufenen Transitions und Nodes jeweils blau dargestellt. Wird ein Breakpunkt erreicht, bleibt die *zen Engine* jeweils vor und nach Ausführung des entsprechenden Bearbeitungsschrittes stehen. Das Breakpunkt-Symbol wird beim ersten Stop links rot umrandet, beim zweiten Stop rechts.

Debug Data Model View

In der View *Debug Data Model* können die Requestdaten, die aktuellen Arbeitsdaten und die Sessiondaten jeweils in einer Baumansicht inspiziert werden. Hierzu wird das Datenmodell komplett dargestellt. Liegen für Elemente des Datenmodells noch keine Anwendungsdaten vor, werden diese mit (n/a) gekennzeichnet. Ansonsten steht hinter den Atomen in Klammern jeweils die Stringrepräsentation des Dateninhalts und, falls vorhanden, der als Java-Objekt konvertierte Dateninhalt. Zusätzlich wird hier die XML-Ausgabe der jeweils letzten Response angezeigt. Die Protokollierung der Debug-Daten erfolgt erst nach der ersten Aktivierung der *Debug Data Model View*.

6 zen Engine Referenz

Zur Implementierung von Geschäftslogik können neben der *Service-API* die *FOM* und *Core API* vom Anwendungsentwickler genutzt werden. Die *zen Engine* stellt zusätzlich eine Reihe von Funktionen zur Verfügung, um die Anwendungsentwicklung zu unterstützen. Mittels der *Hook API* kann eine *zen*-Anwendung in definierten Einsprungpunkten um zusätzlichen technischen Code erweitert werden, der über die Implementierung von Geschäftslogik hinausgeht. Zur Entwicklung von Daten-Schnittstellen ist es nötig, die Eingabe- und Ausgabeformate der Frontend-Backend-Schnittstelle zu verstehen. Um neue Frontends zu entwickeln, lassen sich von der *zen Engine* genutzte Frontend-Java-Klassen erweitern.

6.1 Service-API

Bei der Implementierung der Geschäftslogik werden auch verschiedene technische Services benötigt. Neben den standardisierten Services der EJB-Spezifikation stellt die *zen Platform* zusätzlich eine Reihe von grundlegenden Services wie Logging, Managed Messaging und Datei- bzw. Ressourcenzugriff auch im Applikationsserver zur Verfügung.

Jeder einzelne Service kann ohne Einschränkung aus allen Operations genutzt werden. Die Anwendung von Services über die Service-API der *zen Platform* ist unabhängig vom Deployment einer *zen*-Anwendung, die notwendige technische Anpassung an das Deployment wird ausschließlich über die Konfiguration vorgenommen. Die Service-API abstrahiert von der zugrundeliegenden technischen Basis.

Einmal entwickelter Anwendungscode, der die Service-API nutzt, muß daher auch nicht angepaßt werden, wenn er anfangs in einem Single Cluster Deployment bzw. Servlet-Container gelaufen ist, später aber in einem Applikationsserver eingesetzt wird. Die Skalierung wird durch die Service-API transparent.

Die Konfiguration der einzelnen Services wird ausführlich im Kapitel „Konfiguration der zen Platform“ dokumentiert.

Der Zugriff auf die Service-API erfolgt über die Klasse
de.zeos.zen.scf.service.InitialServiceConnector.
Durch einfaches Instantiieren erhält man eine Instanz des
de.zeos.zen.scf.service.ServiceConnectors,
der die einzelnen Services der Service-API ausprägt.

Beispiel für den Zugriff auf die Service-API. In der Java-Methode
`String ermittleWertpapierName(Integer wkn)`
muß über JDBC auf die Datenbank zugegriffen werden. Dazu könnte die Methode so aussehen:

```
static String ermittleWertpapierName(Integer wkn)
{
    String name = null;
    try {
        ServiceConnector sc = new InitialServiceConnector();
        ConnectionService() cs = sc.getConnectionService();
        javax.sql.Connection con = cs.getConnection(„myConfiguredConnection“);
        // Abfrage der Daten auf 'con' analog zur JDBC-Spezifikation und
        // anschließend Beenden der Verbindung
        return name;
    } catch (ServiceException e) {
        // handle service exception
    } catch (SQLException e) {
        // handle sql exception
    }
}
```

Über den ServiceConnector kann man einzelne Services abrufen. Jeder Service wird durch sein Service-Interface aus dem Package
de.zeos.scf.service
dargestellt. Jeder Service-Zugriff kann eine *ServiceException* auslösen, die ihren Ursprung in einer fehlerhaften Konfiguration oder beispielsweise in Zugriffsproblemen haben kann. Die *ServiceException* wird im weiteren nicht mehr explizit bei jeder API-Methode erwähnt.

6.1.1 Connection

Der *ConnectionService* stellt Verbindungen auf beliebige Datenbanken zur Verfügung. Grundlage dieses Services ist die JDBC-Spezifikation. Der Zugriff erfolgt über die Klasse

`public interface ConnectionService`

- **`public DataSource getConnectionFactory(String poolName)`**: Liefert die JDBC-Datenquelle *poolName*.
- **`public Connection getConnection(String poolName)`**: Liefert direkt eine neue *javax.sql.Connection* für die Datenquelle *poolName*.

Die verwendeten Connection-Pools müssen (zur Laufzeit) in der Konfiguration definiert sein.

Die genaue Dokumentation zur Verwendung der JDBC-API finden Sie Online unter java.sun.com, unter anderem in der JDBC-Spezifikation.

Um transaktionale XAConnections zu nutzen, muß man die entsprechenden Einstellungen in der Konfiguration vornehmen. (->siehe auch dort)

6.1.2 JDO

Der *JDOService* stellt den objektorientierten Zugriff auf die Persistenzschicht zur Verfügung. Grundlage dieses Services ist die *Java Data Objects Specification* (JDO). Die *zen Platform* nutzt den *JDOService*, um auf das Repository der einzelnen *zen*-Anwendungen zuzugreifen. Es kann aber ebenso im Rahmen der Geschäftslogik für objektrelationale Persistenz oder direkt für Objektpersistenz auf objektorientierten Datenbanken verwendet werden. Der Zugriff erfolgt über die Klasse

`public interface JDOService`

- **`public PersistenceManagerFactory getPersistenceManagerFactory (String poolName)`**: Liefert die Factory für neue *PersistenceManager* vom Typ *poolName*.
- **`public PersistenceManager getPersistenceManager(String poolName)`**: Liefert direkt einen neuen *javax.jdo.PersistenceManager* für die Factory *poolName*.

Die verwendeten JDO-Pools müssen (zur Laufzeit) in der Konfiguration definiert sein.

Dokumentation zu den JDO-Implementierungen, die von der zen Platform unterstützt werden, finden Sie im Kapitel „Konfiguration der zen Platform".
Die genaue Dokumentation zur Verwendung der JDO-API finden Sie Online unter java.sun.com, unter anderem in der Java Data Objects Specification.

Sofern Sie eigenen Code zur Objektpersistenz nach der JDO-Spezifikation verwenden, fügen Sie die entsprechenden Klassen einfach dem Klassenpfad beim Aufruf der zen Engine hinzu. Außerdem müssen Sie einen JDO-Pool und einen passenden Connection-Pool in der Konfiguration definieren.

Beispiel für Abfrage von über JDO; die Fehlerbehandlung wurde aus Gründen der Übersichtlichkeit weggelassen:

```
ServiceConnector sc = new InitialServiceConnector();
JDOService js = sc.getJDOService();
PersistenceManager pm = js.getPersistenceManager(MYPM);
Query query = pm.newQuery(BizClass.class);
Collection col = query.execute();
    // Auslesen der Ergebnisse...
query.close();
```

6.1.3 Messaging

Der *MessagingService* bietet nicht nur Unterstützung für Fire-and-Forget-Messages, sondern auch für überwachte Managed Messages. Dadurch können asynchrone Prozesse partiell synchronisiert und überwacht werden. Grundlage dieses Services ist der *Java Message Service* (JMS).

Der Zugriff auf den *MessageService* erfolgt über die Klasse

`public interface MessagingService`

- **`public Messenger getMessenger(String msgName)`**: Liefert eine neue Messenger-Instanz für msg*Name*.
- **`public ManagedMessenger getManagedMessenger(String msgName)`**: Liefert die Instanz des Managed Messengers für *msgName* in der jeweils aktuellen Session. Pro Session existiert maximal eine Instanz für jeden konfigurierten Managed Messenger, d.h. die erste Anfrage in einer Session liefert eine neue Instanz, folgende Anfragen innerhalb der gleichen Session liefern immer die gleiche Instanz zurück, sofern dieser Managed Messenger nicht zwischendurch explizit aufgeräumt wurde. Ebenso erhält man innerhalb der gleichen Session auch nach einem- oder mehreren Request-Response-Zyklen immer den gleichen Managed Messenger im jeweiligen Zustand.

Jeder verwendete Messenger muß (zur Laufzeit) in der Konfiguration definiert sein. Die message-spezifischen Service-Klassen befinden sich alle im Package
de.zeos.scf.service.msg.

Die genaue Dokumentation zur Verwendung der Messaging-API finden Sie Online unter java.sun.com, speziell im Rahmen des Java Message Service (JMS).

➢ *Fire-And-Forget*

Eine Fire-And-Forget-Message wird über den *Messenger* abgesetzt. Als Message Consumer wird hier ein Message Bean erwartet, das zumindest *javax.jms.MessageListener* implementiert. Der Messenger stellt die Message über die konfigurerte Message Queue an das Message Bean zu. Die Message wird anschließend, analog zur EJB-Spezifikation, vom Message Bean in der Methode
`public void onMessage(javax.jms.Message msg)`
asynchron abgearbeitet. Das entsprechende Interface zum Versand von Fire-And-Forget-Messages ist folgendes:

`public interface de.zeos.scf.service.msg.Messenger`

- **`send(Serializable data)`** : Damit werden beliebige Daten an das konfigurierte Standard-Message-Bean versandt. Das Message Bean muß die Daten natürlich geeignet interpretieren können. Als Datencontainer dient intern die *javax.jms.ObjectMessage,* daher ist die einzige Restriktion auf den Daten, daß sie serialisierbar sein müssen.

- **`send(Serializable data, boolean logging)`** : Diese Methode ist nur in äußerst seltenen Fällen notwendig. Sie hat die gleiche Funktionalität wie oben, bietet aber zusätzlich die Möglichkeit, das System-Logging auszuschalten. Dies ist nur notwendig, wenn man Logging-Klassen schreibt, die zur Logausgabe Messaging-Techniken verwenden. Durch Abschalten des System-Loggings werden Logging-Deadlocks verhindert.

Die Verknüpfung von Messenger mit dem entsprechenden Message Consumer finden Sie im Kapitel „Konfiguration der zen Platform“.

➢ Managed Message

Eine *Managed Message* wird über den *ManagedMessenger* abgesetzt. Pro Session existiert jeder konfigurierte *ManagedMessenger* maximal einmal. Als Message Consumer wird hier ein Message Bean erwartet, daß von der Klasse

de.zeos.scf.component.ManagedMessageDefImpl

erben muß. Dieses Bean kann während der Abarbeitung der Message seinen Zustand über vererbte Methoden kommunizieren. Der jeweilige Zustand wird intern an den *ManagedMessenger* weitergeleitet. Der Initiator der Managed Message kann den Abarbeitungszustand laufend abfragen, auch wenn die Abfragen über mehrere Request-Response-Zyklen verteilt sind, da er pro Session immer die gleiche Instanz erhält.

siehe dazu auch das Beispiel im Kapitel „Patterns“.

Das entsprechende Interface für Managed Messages ist folgendes:

`public interface de.zeos.scf.service.msg.ManagedMessenger`

- **`public void setMessageData(Map map)`**: Setzt die Daten, die das konfigurierte Message Bean zur (asynchronen) Abarbeitung benötigt. Das Bean erhält später exakt die gleiche Map, muß die Werte also geeignet interpretieren können.

- **`public void startMessaging()`**: Startet den Prozeß des Managed Messagings. Da Messages immer asynchron abgearbeitet werden, kehrt die Methode sofort zurück.

- **`public boolean messagingStarted()`**: Liefert, ob dieser *ManagedMessenger* zuvor schon gestartet wurde.

- **`public boolean messagingFinished()`**: Liefert, ob dieser *ManagerMessenger* inzwischen seine Arbeit beendet hat. Ein *ManagedMessenger* kann auch mit Fehlern beendet werden.

- **`public MessageStatus getStatus()`**: Liefert das Status-Objekt, das den aktuellen Zustand des *ManagedMessengers* beschreibt.

- **`public Object getErrorObject()`**: Liefert ein eventuell vom Message Bean gesetztes beliebiges Fehlerobjekt.

- **`public void cleanUp()`**: Räumt diesen *ManagedMessenger* für die aktuelle Session auf. Solange ein einmal gestarteter *ManagedMessager* nicht aufgeräumt wurde, erhält man bei Anfragen nach dem *ManagedMessenger* gleichen Namens immer die gleiche, schon laufende Instanz. Erst das Aufräumen sorgt dafür, daß eventuell folgende Zugriffe innerhalb der gleichen Session auf den *ManagedMessenger* gleichen Namens eine neue Instanz liefern.

6.1.4 Transaction

Der *TransactionService* sorgt für Datenkonsistenz auf transaktionalen Ressourcen. Grundlage dieses Services ist die *Java Transaction API* (JTA). Momentan muß hier zwischen Single Container - und Cluster Deployment unterschieden werden:

- **Single Container Deployment**: Sofern man einen *ConnectionService*-Pool als transaktional kennzeichnet, wird die Connection als XAConnection geladen und diese zum Transaktionsmanager hinzugefügt. Andere Ressourcen werden bei diesem Deployment momentan noch nicht vom Transaktionsmanager verwaltet. (->siehe auch *Konfiguration*)

- **Cluster Deployment**: Hier hängt die Menge der Ressourcen, die vom Transaktionsmanager verwaltet werden, von der Implementierung und Konfiguration des jeweiligen Applikationsservers ab.

Demnächst werden auch bei Single Container Deployment weitere transaktionale Ressourcen verfügbar sein.

Der Zugriff erfolgt über die Klasse:

`public interface TransactionService`

- **`public UserTransaction getUserTransaction()`**: Liefert die jeweils aktuelle *UserTransaction* auf der man eine Transaktion beginnen und abschließen oder zurücksetzen kann.

Beispiel für die Verwendung des TransactionService als Klammer über eine Connection und eine Nachricht; die Fehlerbehandlung wurde aus Gründen der Übersichtlichkeit weggelassen:

```
ServiceConnector sc = new InitialServiceConnector();
TransactionService ts = sc.getTransactionService();
UserTransaction utx = ts.getUserTransaction();
utx.begin();
    Connection con = sc.getConnectionService().getConnection();
        // Anwendung der Connection für Datenbankupdates o.ä.
    con.commit();
    con.close();
    // Senden einer Message
    Messenger msg = sc.getMessagingService.getMessenger(MSG);
    msg.send(„A message“);
utx.commit();
```

Die genaue Dokumentation zur Verwendung der java Transaction API finden Sie Online unter java.sun.com.

6.1.5 Logging

Über den *LoggingService* kann man aus der Geschäftslogik heraus Meldungen transparent in beliebige Ausgabekanäle senden. Jede Meldung wird dabei mit einem Level versehen, das die Art bzw. Wichtigkeit dieser Meldung beschreibt. Grundlage dieses Services bezüglich der Anwendungsentwicklung ist *Jakarta Commons Logging*. Als Log-Subsysteme werden momentan *Log4J* und die *Java-Logging-API* unterstützt.

Dokumentation zur Konfiguration der einzelnen Logger finden Sie im Kapitel „Konfiguration der zen Platform“. Dort wird dokumentiert, wie man die einzelnen Ausgabekanäle definiert und die Ausgabe auf vorgegebene Log-Level einschränkt.

Der Zugriff erfolgt über die Klasse:

`public interface LoggingService`

- **`public Log getLogger(String name)`**: Liefert den Logger mit der Bezeichnung *name*.
- **`public Log getLogger(String name, boolean warn)`**: Wie oben, sofern der entsprechende Logger aber nicht explizit konfiguriert ist, wird eine Meldung ausgegeben.
- **`public Log getFallbackLogger()`**: Liefert einen Logger, der in jedem Fall funktioniert. Dieser Logger sollte angewandt werden, sofern ein anderer fehlerhaft arbeitet, beispielsweise weil dessen Logkanal (z.B. die Datenbank) nicht erreichbar ist.

Beispiel für die Verwendung des LoggingService; die Fehlerbehandlung wurde aus Gründen der Übersichtlichkeit weggelassen:

```
ServiceConnector sc = new InitialServiceConnector();
LoggingService ls = sc.getLogService();
Log log = ls.getLogger(„my.company“);
log.info(„This is log output“);
```

Das Ausgabeziel wird durch die Konfiguration festgelegt. Der Logeintrag kann also ebenso in der Datenbank oder in einer Datei abgelegt und/oder zusätzlich als Mail versandt werden.

Ein Logger muß, anders als beispielsweise eine Connection, nicht geschlossen werden. Die Reihenfolge der Ausgabe ist durch die Reihenfolge der Aufrufe festgelegt, auch wenn der gleiche Logger von mehreren Stellen aus genutzt wird.

Die genaue Dokumentation zur Verwendung von Jakarta Commons Logging finden Sie Online unter jakarta.apache.org.

6.1.6 Mail

Der *MailService* kapselt den Zugriff auf den Mail-Server. Grundlage dieses Services ist die *JavaMail API*. Der Zugriff erfolgt über die Klasse:

`public interface MailService`

- **`Session getMailSession()`**: Liefert die *javax.mail.Session*, die den Zugriff auf den Mail-Server ermöglicht.
- **`MailHandler getMailHandler()`**: Liefert einen jeweils neue *MailHandler*-Instanz.

Die genaue Dokumentation zur Verwendung der JavaMail API finden Sie Online unter java.sun.com.

Ein *MailHandler* stellt der Anwendungsschicht optimierte Funktionalität auf Basis der Mail-Session zur Verfügung, beispielsweise um Mails mit Attachments zu erstellen. Der *MailHandler* stellt dazu folgende Möglichkeiten bereit:

de.zeos.scf.service.mail.MailHandler

- **`public void setSender(String sender)`**: Setzt den Sender der Mail. Der Sender muß eine korrekte Mailadresse sein.

- **public void setSender(InternetAddress sender)**: Setzt ebenso den Sender der Mail.
- **public void setRecipients(String [] recipients)**: Setzt mehrere Empfänger auf einmal. Jeder Empfänger muß eine korrekte Mailadresse sein.
- **public void setRecipients(InternetAddress [] recipients)**: Setzt ebenso mehrere Empfänger auf einmal.
- **public void setSubject(String subject)**: Setzt den Betreff der Mail.
- **public void setContent(String content)**: Setzt den eigentlichen Inhalt der Mail.
- **public void setContent(File file)**: Setzt ebenso den Inhalt der Mail, hier wird aber die angegebene Datei ausgelesen. Achtung: Diese Operation kann analog zur J2EE-Spezifikation nur im Frontend funktionieren!
- **public void addPDF(String filename, byte [] data)**: Fügt das PDF *data* als Attachment mit Namen *filename* an.
- **public void addText(String filename, String text)**: Fügt den Text *text* als Attachment mit Namen *filename* an.
- **public void addAttachment(BodyPart part)**: Jeder Aufruf dieser Method fügt ein weiteres *BodyPart*-Attachment an die Mail. Der *BodyPart* muß in diesem Fall extern erstellt werden, der *MimeType* ist dafür nicht auf PDF oder Text beschränkt.
- **public void send()**: Sendet die Mail.
- **public void clear()**: Setzt den internen Zustand des *MailHandlers* zurück.

Beispiel für die Verwendung des MailService; die Fehlerbehandlung wurde aus Gründen der Übersichtlichkeit weggelassen:

```
ServiceConnector sc = new InitialServiceConnector();
MailService ms = sc.getMailService();
MailHandler mh = ms.getMailHandler();
mh.setSubject(„Example“);
mh.setSender(„info@zeos.de“);
mh.setRecipients(new String [] {„your@mail.address“});
mh.setContent(„This is an example“);
mh.addText(„useThisFileName.txt“, „This is the attached file content“);
mh.send();
```

6.1.7 SessionManagement

Über den *SessionManagerService* kann direkt auf die aktuelle Session zugegriffen werden. In der Session können beliebige Daten abgelegt werden, sofern neben dem *Data Model* einer Anwendung weitere Daten gespeichert werden müssen. Das Sessionmanagement kann sich automatisch mit anderen Rechnern synchronisieren, wenn die *zen Platform* in einem Cluster läuft. Der Zugriff erfolgt über die Klasse:

public interface SessionManagerService

- **public SessionManager getSessionManager()**: Liefert den *SessionManager* für den Zugriff auf die Sessiondaten.

Über den *SessionManager* kann man auf die jeweils gültige *SCFSession* zugreifen:

de.zeos.scf.service.sm.SessionManager

- **public SCFSession getSession()**

Auf der *SCFSession* kann man Sessiondaten lesen und schreiben. Dazu stehen folgende Möglichkeiten bereit:

de.zeos.scf.service.sm.SCFSession

- **public void putValue(String key, Object obj)**: Legt *obj* unter dem Schlüssel *key* in der Session ab. Eventuell vorhandene Werte mit gleichem Schlüssel werden überschrieben, *obj* darf nicht *null* sein.
- **public Object getValue(String key)**: Liefert die Daten, die zuvor unter dem Schlüssel *key* in der Session abgelegt wurden.
- **public void removeValue(String key)**: Entfernt die Daten, die mit dem Schlüssel *key* verknüpft sind, aus der Session.
- **public long getCreationTime():** Liefert den Zeitpunkt, an dem diese Session erstellt wurde, in Millisekunden seit Mitternacht, 1. Januar 1970 (UTC).

6.1.8 ResourceRepository

Das *ResourceRepository* dient zum lesenden und schreibenden Zugriff auf beliebige Text- oder Binär-Ressourcen (Es hat nichts mit den Ressourcen einer *zen*-Anwendung zu tun, die in der *Resource View* erstellt werden). Die Quelle der Ressourcen kann das Dateisystem oder ein Http-Server sein, es können aber auch neue Quellen wie beispielsweise eine Datenbank entwickelt werden. Der Zugriff auf die Ressourcen bzw. die tatsächliche Datenquelle bleibt aus Sicht der Anwendungsentwicklung völlig transparent.

Dokumentation zur Konfiguration der einzelnen ResourceRepositories abhängig vom Deployment finden Sie im Kapitel „Konfiguration der zen Platform".

Der Zugriff erfolgt über die Klasse:

`public interface ResourceRepositoryService`

- **`ResourceRepository getResourceRepository(String name)`**: Liefert das *ResourceRepository* mit dem Namen *name*.
- **`Set getResourceRepositoryNames()`**: Liefert ein read-only Set mit den Namen aller definierten *RessourceRepositories*.

Auf jedem *ResourceRepository* kann man Ressourcen lesen. Ob Schreiben möglich ist, hängt jeweils von der Konfiguration ab. Der Zugriffspfad auf eine Ressource ist immer relativ zur Wurzel des *ResourceRepositories* zu sehen. Daher wird der Anwendungscode auch nicht tangiert, wenn die Datenquelle in der Konfiguration verschoben wird oder eine völlig andere Datenquelle konfiguriert wird.

`de.zeos.scf.service.res.ResourceRepository`

- **`public String getTextResource(String relPath)`**: Liefert die Textressource, die unter dem relativen Pfad *relPath* liegen muß.
- **`public byte [] getBinaryResource(String relPath)`**: Wie oben, in diesem Fall wird aber eine Binärressource geliefert.
- **`public void putTextResource(String relPath, String text, boolean dirAutoCreate)`**: Legt den String *text* unter dem relativen Pfad *relPath* unter der Wurzel des *ResourceRepository* ab. Sofern *dirAutoCreate true* ist, werden notwendige neue Verzeichnissstrukturen automatisch angelegt. Ist es *false* und die notwendigen Verzeichnisstrukturen existieren nicht, wird eine *ServiceException* geworfen.
- **`public void putBinaryResource(String relPath, byte [] data, boolean dirAutoCreate)`**: Wie oben, in diesem Fall wird aber eine Binärressource geschrieben.
- **`public void createDirectory(String relPath)`**: Erstellt das Verzeichnis *relPath* relativ zur Wurzel des *ResourceRepositories*.

Beispiel für die Verwendung eines dateisystembasierten ResourceRepository, die Wurzel ist konfiguriert als „file:/opt/"; die Fehlerbehandlung wurde aus Gründen der Übersichtlichkeit weggelassen:

```
ServiceConnector sc = new InitialServiceConnector();
ResourceRepositoryService rrs = sc.getResourceRepositoryService();
ResourceRepository rr = rrs.getResourceRepository(„myName");
// lesen der Datei file:/opt/myfile.txt
String content = rr.getTextResource(„myfile.txt");
// lesen der PDF-Datei file:/opt/data/another.pdf
byte [] pdfData = rr.getBinaryResource(„/data/another.pdf");
// schreiben der Datei file:/opt/newFile
rr.putTextResource(„newFile", „Content of new file", true);
```

Exakt der gleiche Code würde genauso funktionieren, wenn ein Http-Server-basiertes ResourceRepository konfiguriert wäre, unter dessen Wurzel die gleiche Datenstruktur liegt. Der Http-Server muß dabei die PUT-Methode der Spezifikation unterstützen, um Schreibzugriff zu ermöglichen.

Durch die Verwendung eines Http-Servers als *ResourceRepository* werden auch im Applikationsserver Dateizugriffe möglich, ohne die EJB-Spezifikation zu verletzen.

6.1.9 ComponentSelector

Über den *ComponentSelectorService* kann man transparent auf SCF-Komponenten zugreifen. Das Deployment einer SCF-Komponente ist daher flexibel und weitgehend unabhängig vom Aufruf der Komponente. Die einzige Einschränkung dabei ist, daß aus dem Backend nicht auf Frontend-Komponenten zugegriffen werden kann. Der Zugriff auf den *ComponentSelector* erfolgt über die Klasse:

`public interface ComponentSelectorService`

- **`public ComponentSelector getComponentSelector()`**: Liefert den *ComponentSelector* zum Zugriff auf SCF-Komponenten.

Über den *ComponentSelector* kann man auf die Komponenten zugreifen:

`public interface ComponentSelector`

- **`public Component getComponent()`**: Liefert die SCF-Komponente, die als *name* konfiguriert wurde.

6.2 FOM API

Die *FOM API* ist eine Schnittstelle aus dem Archiv *zenapi.jar* (Package *de.zeos.zen.api.fom*), die es Operationen erlaubt, direkt auf modellierte Anwendungsdaten zuzugreifen. Im Modell einer *zen*-Anwendung können einer Operation oder einer Domäne Element-Argumente aus dem Datenmodell zugewiesen werden, die als *call by reference* deklariert sind. In diesen Fällen erfolgt der Zugriff auf die Anwendungsdaten zur Laufzeit über die *FOM API*. Diese

ermöglicht es, über die gesamten Anwendungsdaten zu iterieren und die Laufzeitstruktur der Daten-Ausprägung zu verändern. Das eigentliche Modell der Anwendungsdaten bleibt davon unberührt. Es liegt in der Verantwortung des Anwendungsentwicklers, die Konsequenzen von Strukturänderungen zu beachten, wenn sie inkonsistent zum Modell sind.

6.2.1 Element

Das *Element* ist die abstrakte Vaterklasse der konkreten FOM-Elemente *Atom*, *Composition* und *List*. Sie beinhaltet Methoden, die unabhängig vom konkreten FOM-Element sind.

`public interface Element`

- **`public String getName()`:** Liefert den Namen des FOM-Elements zurück
- **`public Type getType()`:** Liefert den Typ des FOM-Elements zurück. Dieser kann mit dem Operator == verglichen werden mit:
  - *`Element.Type.ATOM`*
  - *`Element.Type.LIST`*
  - *`Element.Type.COMP`*
- **`public void setAttribute(String name, String value)`:** Setzt ein Attribut auf das FOM-Element mit dem Namen *name* und dem Wert *value*. Existiert bereits ein Attribut mit dem angegebenen Namen auf dem Element, wird es überschrieben. Attribute sind Bestandteil bei der Ein- und Ausgabe von FOM-Elementen und können z.B. bei in Stylesheets genutzt werden. Diese Methode wirft eine *FOMRuntimeException*, wenn folgende Parameter übergeben werden:
  - *name* ist leer ("") oder *null*
  - *name* ist ungültig, d.h. entspricht nicht dem regulären Ausdruck
    `letter(letter|digit|'_'|'–'|'.'|':')*`
    Mit einem Doppelpunkt werden Namespaces vom eigentlichen Attributsnamen abgetrennt. Bis auf den Namespace *builtin* mit fest definierten Attributen dürfen derzeit keine weiteren Namespaces verwendet werden.
  - *value* ist *null*
- **`public String getAttribute(String name)`:** Liefert den Wert des Attributes mit dem angegebenen Namen zurück. Existiert kein Attribut mit dem angegebenen Namen auf dem FOM-Element, wird *null* zurückgegeben
- **`public String removeAttribute(String name)`:** Löscht das Attribut mit dem angegebenen Namen und liefert den Wert des Attributs zurück. Existiert kein Attribut mit dem angegebenen Namen auf dem FOM-Element, wird *null* zurückgegeben.
- **`public Iterator getAttributeIterator()`:** Liefert einen Iterator über alle Attribute des FOM-Elements zurück.
- **`public Element getParent()`:** Gibt das Vaterelement des aktuellen FOM-Elements zurück. Hat das aktuelle FOM-Element kein Vaterelement, wird *null* zurückgegeben.
- **`public Element getRoot()`:** Gibt das Wurzelelement des FOM-Baumes zurück, zu dem das aktuelle FOM-Element gehört. Liefert das aktuelle FOM-Element zurück, wenn es selbst das Wurzelelement des FOM-Baumes ist.
- **`public String toXML()`:** Liefert eine XML-Repräsentation des aktuellen FOM-Elements mit allen Kindelementen zurück.
- **`public Element deepClone()`:** Liefert einen Klon des aktuellen FOM-Elements mit Attributen und Dateninhalten über alle Kindelemente zurück. Da bei Atomen die Dateninhalte mitgeklont werden, sollten die Datentypen das Interface *Cloneable* implementieren. Anderenfalls könnten Dateninhalte durch den Klonvorgang auf *null* gesetzt werden oder ihre Java-Referenz mit der Klonvorlage teilen.
- **`public Element flatClone()`:** Liefert einen Klon des aktuellen FOM-Elements mit Attributen und Dateninhalten, aber ohne Kindelemente zurück. Da bei Atomen die Dateninhalte mitgeklont werden, sollten die Datentypen das Interface *Cloneable* implementieren. Anderenfalls könnten Dateninhalte durch den Klonvorgang auf *null* gesetzt werden oder ihre Java-Referenz mit der Klonvorlage teilen.

6.2.2 Atom

Ein *Atom* ist ein FOM-Element, das nur als Blatt eines FOM-Baumes vorkommen kann. Es hat insbesondere einen Dateninhalt. Bei Atomen, denen Benutzereingaben zugrunde liegen, wurde der Dateninhalt durch die *zen Engine* über den entsprechenden Datentyp-Konvertierer gefüllt.

`public interface Atom extends Element`

- **`public Object getData()`:** Liefert den Dateninhalt des Atoms zurück
- **`public void setData(Object data)`:** Setzt den Dateninhalt des Atoms

Jedes Atom läßt sich zur Laufzeit auf das Interface *SourceAtom* casten. Damit ist nicht nur der sprachunabhängige Objekt-Dateninhalt des Atoms zugreifbar, sondern auch die originale sprachspezifische Stringrepräsentation des Dateninhaltes. Normalerweise ist der Zugriff auf diesen durch den Anwendungsprogrammierer nicht nötig.

`public interface SourceAtom extends Atom`

- **`public String getSourceValue():`** Liefert die Stringrepräsentation des Dateninhalts des Atoms zurück
- **`public void setSourceValue(String value):`** Setzt die Stringrepräsentation des Dateninhalt des Atoms

6.2.3 ElementCollection

Eine *ElementCollection* ist ein Interface, das gemeinsame Eigenschaften einer Composition und einer Liste zusammenfaßt.

`public interface ElementCollection extends Element`

- **`public Iterator iterator():`** Liefert einen Iterator über alle Kindelemente der Composition oder Liste zurück.
- **`public int size():`** Liefert die Anzahl der Kindelemente der Composition oder Liste zurück.

6.2.4 Composition

Eine *Composition* ist ein FOM-Element, das verschiedenartige FOM-Elemente als Kinder haben kann. Diese müssen insbesondere auch verschiedene Namen besitzen.

`public interface Composition extends ElementCollection`

- **`public Element getElement(String name):`** Liefert das Kind der Composition mit dem angegebenen Namen zurück. Existiert kein Kind mit dem angegebenen Namen, wird *null* zurückgegeben.
- **`public Element setElement(Element element):`** Setzt das angegebene Element als Kind in die aktuelle Composition. Liefert ein gleichnamiges Kind zurück, das zuvor der Composition zugeordnet war, oder *null*, wenn die Composition kein gleichnamiges Kind besaß. Diese Methode wirft eine *FOMRuntimeException*, wenn folgende Situationen eintreten:
    - *element* ist bereits einem anderen *Collection-Element* als Kind zugeordnet
    - *element* ist das Wurzelelement der aktuellen Composition
- **`public void removeElement(String name):`** Löscht das Element mit dem angegebenen Namen aus der Composition. Diese Methode wirft eine *FOMRuntimeException*, wenn kein Kind mit dem angegebenen Namen existiert.
- **`public Element disconnectElement(String name):`** Hängt das Element mit dem angegebenen Namen aus der Composition. Liefert das ausgehängte Element zurück. Diese Methode wird verwendet, wenn Teilbäume innerhalb eines FOM-Baumes an eine andere Stelle gehängt werden sollen. Diese Methode wirft eine *FOMRuntimeException*, wenn kein Kind mit dem angegebenen Namen existiert.

6.2.5 List

Eine *List* ist ein FOM-Element, das nur gleichartige FOM-Elemente als Kinder haben kann. Diese dürfen sich insbesondere nicht im Namen unterscheiden.

`public interface List extends ElementCollection`

- **`public Element getElement(int i):`** Liefert das i-te Kind der Liste zurück. Wirft eine *FOMRuntimeException*, wenn kein Kind mit dem angegebenen Index existiert
- **`public void addElement(int i, Element element):`** Fügt das angegebene Element der Liste an i-ter Position hinzu. Die Indexe der nachfolgenden Elemente, falls vorhanden, werden inkrementiert. Ist der angegebene Index höher als die aktuelle Listengröße, werden entsprechend viele gleichartige Dummy-Elemente (ohne Inhalt und Kindern) erzeugt und der Liste hinzugefügt. Diese Methode wirft eine *FOMRuntimeException*, wenn folgende Situationen eintreten:
    - *element* ist bereits einem anderen *Collection-Element* als Kind zugeordnet
    - *element* ist das Wurzelelement der aktuellen Liste
    - *element* hat einen anderen Namen als die anderen Listenelemente
    - $i < 0$
- **`public void addElement(Element element):`** Fügt das angegebene Element der Liste an letzter Position hinzu. Diese Methode wirft eine `FOMRuntimeException`, wenn folgende Situationen eintreten:
    - *element* ist bereits einem anderen *Collection-Element* als Kind zugeordnet

- *element* ist das Wurzelelement der aktuellen Liste
- *element* hat einen anderen Namen als die anderen Listenelemente

- **`public void removeElement(int index):`** Löscht das i-te Element aus der Liste. Die Indexe der nachfolgenden Elemente ändern sich entsprechend. Diese Methode wirft eine *FOMRuntimeException*, wenn kein Element mit dem angegebenen Index existiert.
- **`public Element disconnectElement(int index):`** Hängt das i-te Element aus der Liste. Liefert das ausgehängte Element zurück. Diese Methode wird verwendet, wenn Teilbäume innerhalb eines FOM-Baumes an eine andere Stelle gehängt werden sollen. Diese Methode wirft eine *FOMRuntimeException*, wenn kein Element mit dem angegebenen Index existiert.
- **`public int indexOf(Element element):`** Liefert den Index des angegebenen Elements zurück, oder -1, wenn das angegebene Element nicht in der Liste enthalten ist.

6.2.6 ElementBuilder

Da FOM-Elemente als Interfaces vorliegen, können nicht direkt instantiiert werden. Hierzu dient die Klasse *ElementBuilder.*

`public abstract class ElementBuilder`

- **`public static synchronized ElementBuilder getInstance():`** Liefert die (thread-sichere) Singleton-Instanz eines *ElementBuilder* zurück. Wirft eine *FOMRuntimeException*, wenn das Archiv *zencore.jar* nicht im Classpath enthalten ist.
- **`public abstract Atom newAtom(String name):`** Instantiiert ein Atom mit dem angegebenen Namen. Diese Methode wirft eine *FOMRuntimeException*, wenn folgende Situationen eintreten:
  - *name* ist leer ("") oder *null*
  - *name* ist ungültig, d.h. entspricht nicht dem regulären Ausdruck
    `letter(letter|digit|'_'|'–'|'.'|':')*`
    Mit einem Doppelpunkt werden Namespaces vom eigentlichen Elementnamen abgetrennt. Derzeit dürfen jedoch keine Namespaces verwendet werden.
- **`public abstract Composition newComposition(String name):`** Instantiiert eine Composition mit dem angegebenen Namen. Diese Methode wirft eine `FOMRuntimeException`, wenn folgende Situationen eintreten:
  - *name* ist leer ("") oder *null*
  - *name* ist ungültig, d.h. entspricht nicht dem regulären Ausdruck
    `letter(letter|digit|'_'|'–'|'.'|':')*`
    Mit einem Doppelpunkt werden Namespaces vom eigentlichen Elementnamen abgetrennt. Derzeit dürfen jedoch keine Namespaces verwendet werden.
- **`public abstract List newList(String name):`** Instantiiert eine Liste mit dem angegebenen Namen. Diese Methode wirft eine `FOMRuntimeException`, wenn folgende Situationen eintreten:
  - *name* ist leer ("") oder *null*
  - *name* ist ungültig, d.h. entspricht nicht dem regulären Ausdruck
    `letter(letter|digit|'_'|'–'|'.'|':')*`
    Mit einem Doppelpunkt werden Namespaces vom eigentlichen Elementnamen abgetrennt. Derzeit dürfen jedoch keine Namespaces verwendet werden.

6.2.7 Path

Diese Klasse dient dazu, FOM-Elemente in einem FOM-Baum mit stringbasierten Pfadausdrücken zu finden bzw. FOM-Elemente anhand eines Pfadausdruckes zu konstruieren. Die Pfadausdrücke orientieren sich an XPATH und werden wie folgt definiert:

- `Path ::= RelPath|AbsPath`
- `AbsPath ::= '/' RelPath?`
- `RelPath ::= Step | RelPath '/' Step`
- `Step ::= AbbrStep | TypeIdent? Name Predicate?`
- `AbbrStep ::= '..'`
- `TypeIdent ::= '$'`
- `Name ::= letter (letter | digit | '_' | '-' | '.' | ':')*`
- `Predicate ::= '[' Expr ']'`
- `Expr ::= Index | AndExpr`

- `Index ::= digit+`
- `AndExpr ::= AttrExpr 'and' AndExpr | AttrExpr`
- `AttrExpr ::= '@' Name '=' ''' char* '''`

Ein Pfadausdruck beschreibt den Weg von einem Kontext-Element zu einem Element unterhalb des Kontext-Elements, indem die Namen der jeweiligen Kind-Elemente durch `'/'` getrennt aneinandergereiht werden. Relative Pfadausdrücke sind relativ zum Kontext-Element und beginnen mit dem Namen des Kindelements, absolute Pfade beginnen mit `'/'` und der Angabe des Wurzelelements des FOM-Baumes. Mit `'..'` wird der Kontext auf das Vaterelement des aktuellen Kontext-Elements gesetzt. Der Typbezeichner `'$'` wird verwendet, um Listen von Compositions zu unterscheiden. Dieser braucht jedoch nur beim Konstruieren von Elementen mittels Pfadausdrücken angegeben werden. Mit Angabe eines Prädikats kann einerseits das i-te Element einer Liste identifiziert werden, andererseits ein Element mit einem bestimmten Attributwert gesucht werden.

Der absolute Pfadausdruck `/data/bar/com` findet vom Wurzelelement `data` aus das Element `com` unterhalb des Elements `bar`
Der relative Pfadausdruck `bar/com` findet vom aktuellen Kontext-Element aus das Element `com` unterhalb des Kindelements des Kontext-Elements `bar`
Der Pfadausdruck `../foo` findet das Geschwisterelement `foo` des aktuellen Kontext-Elements
Der Pfadausdruck `$liste/listenelement[i]` findet das i-te Element `listenelement` der Liste `liste`
Der Pfadausdruck `comp/compelement[@foo='bar']` sucht nach einem Element `compelement` unter einer Composition `comp`, dessen Attribut names `foo` den Wert `bar` hat.

`public abstract class Path`

- **`public static Path newPath(Element element)`**: Liefert eine `Path`-Instanz mit dem angegebenen Element als Kontext-Element und einer maximalen Listengröße von 100 zurück, wenn Elemente mit dieser Instanz erzeugt werden. Wirft eine *FOMRuntimeException*, wenn das Archiv *zencore.jar* nicht im Classpath enthalten ist.
- **`public static Path newPath(Element element, int maxListSize)`**: Liefert eine `Path`-Instanz mit dem angegebenen Element als Kontext-Element und der angegebenen maximalen Listengröße, wenn Elemente mit dieser Instanz erzeugt werden. Wirft eine *FOMRuntimeException*, wenn das Archiv *zencore.jar* nicht im Classpath enthalten ist.
- **`public abstract Element find(String path)`**: Liefert das Element zurück, das mit dem angegebenen Pfadausdruck vom aktuellen Kontext-Element aus gefunden wird, oder *null*, wenn kein Element unter dem angegebenen Pfadausdruck gefunden werden kann.
- **`public abstract String toString()`**: Liefert die Stringrepräsentation der aktuellen `Path`-Instanz als Pfadausdruck zurück.
- **`public abstract Atom create(String path)`**: Konstruiert ein Atom und integriert dieses an derjenigen Stelle in den FOM-Baum, die durch das Kontext-Element und den Pfadausdruck bestimmt wird. Elemente, die auf dem Weg vom Kontext-Element zum Atom fehlen, werden ebenfalls erstellt. Wird im Pfadausdruck ein Listenelement mit einem Index angegeben, der größer ist als die aktuelle Listengröße, werden zusätzliche Dummy-Elemente erstellt, um die Lücke zu füllen. Diese Dummy-Elemente haben weder Inhalt noch Kindelemente. Dabei darf jedoch die in der `Path`-Instanz angegebene maximale Listengröße nicht überschritten werden. Der Typbezeichner einer Liste muß immer angegeben werden. Der Pfadausdruck muß immer mit einem Atom enden.

Ein FOM-Baum besteht aus der Wurzel `data`*, den Kindelementen* `a` *(Atom) und* `b` *(Composition). Wird nun das Statement* `Path.newPath("b").create("x/y")` *auf dem FOM-Baum aufgerufen, wird unter der Composition* `b` *eine neue Composition* `x` *mit einem Atom* `y` *angelegt.*

6.3 Core API

In der Core API im Archiv *zenapi.jar* (Package *de.zeos.zen.api.core*) befinden sich einige Klassen und Interfaces, die zur Implementierung der Geschäftslogik einer *zen*-Anwendung benötigt werden.

6.3.1 OperationException

Mittels einer *OperationException* bzw. einer *ValidationRuleException* signalisiert eine Operation einen Benutzerfehler. Zur Auswahl einer Fehlermeldung wird die Exception mit einem Fehlerbezeichner versehen, der dem Namen einer zur Operation zugeordneten Fehlermeldung entsprechen muß. Zusätzlich werden alle Elemente als fehlerhaft markiert, die der Operation als Eingabeparameter zugeordnet waren.

`public class OperationException extends ZenException`

- **`public OperationException()`**: Konstruiert eine `OperationException` ohne Fehlerbezeichner. Ist der Operation nur eine einzige Fehlermeldung zugeordnet, wird diese ausgewählt. Ansonsten resultiert ein Anwendungsfehler in der *zen Engine*.

- **`public OperationException(String errLabelName)`:** Konstruiert eine `OperationException` mit dem angegebenen Fehlerbezeichner. Als Benutzerfehler wird diejenige Fehlermeldung ausgewählt, deren Name diesem Bezeichner entspricht. Ist keine solche Fehlermeldung zugeordnet, resultiert ein Anwendungsfehler in der *zen Engine*.
- **`public void setErrorElements(Set errorElements)`:** Definiert eine Menge von Elementen, die die *OperationException* verursacht haben. Defaultmäßig werden alle Eingabeparameter der Operation als fehlerhaft markiert. In bestimmten Fällen kann es jedoch nötig sein, die Fehlermarkierungen an anderen Elementen anzubringen, z.B. wenn diese durch FOM-Navigation von den Eingabe-Elementen der Operation erreicht werden. Mit Angabe einer Menge von FOM-Elementen werden statt der Eingabeparameter die angegebenen FOM-Elemente als fehlerhaft markiert.
- **`public Set getErrorElements()`:** Liefert die Menge der als fehlerhaft markierten FOM-Elemente zurück, die die defaultmäßige Markierung der Eingabeparameter ersetzen soll.
- **`public String getErrLabelName()`:** Liefert den Fehlerbezeichner der *OperationException* zurück.

`public class ValidationRuleException extends OperationException`

Besitzt die gleiche Schnittstelle.

6.3.2 OperationRuntimeException

Mit einer *OperationRuntimeException* kann eine Operation einen kritischen Anwendungsfehler signalisieren. Sie kann dabei einen Fehlertext und ggf. zusätzlich ein Throwable mitgeben, die den Anwendungsfehler ausgelöst hat. Der Fehlertext erscheint in der Log-Ausgabe. Bei kritischen Anwendungsfehlern kann die Bearbeitung des Workflows abgebrochen werden.

`public class OperationRuntimeException extends ZenRuntimeException`

- **`public OperationRuntimeException(String msg)`:** Konstruiert eine `OperationRuntimeException` mit dem übergebenen Fehlertext.
- **`public OperationRuntimeException(String msg, Throwable ex)`:** Konstruiert eine `OperationRuntimeException` mit dem übergebenen Fehlertext und einem Throwable, das den Anwendungsfehler hervorgerufen hat.

6.3.3 AggregationOperation

Wird eine Operation mit der Eigenschaft *aggregation* modelliert, muß ihre Java-Klasse das Interface *AggregationOperation* implementieren. Die Klasse einer solchen Operation wird vor Ausführung instantiiert, ihre Methode entsprechend der Anzahl der als Parameter modellierten Listen-Elemente mehrfach ausgeführt. In der Instanz kann zwischenzeitlich ein Zustand über die mehrfachen Aufrufe hinweg mitgeführt und modifiziert werden. Nach der letzten Ausführung wird die Methode *getResult()* aufgerufen, in der das Ergebnis zurückgeliefert wird. Da pro Operation eine Instanz erzeugt wird, ist es durchaus möglich, daß eine *AggregationOperation*-Klasse mehrere verschiedene Operations enthält, wenn darauf geachtet wird, das jeweils passende Ergebnis zurückzuliefern.

`public interface AggregationOperation`

- **`public Object getResult()`:** Liefert das Ergebnis der Aggregation zurück

6.3.4 Domain

Eine entwicklerdefinierte Domäne implementiert das Interface *Domain*. Sie übernimmt die locale-spezifische Definition der Key/Value-Paare einer Domäne. Die Menge der Keys ist normalerweise für jede Locale gleich und unterscheidet sich bei Aufzählung nur in der Reihenfolge. Es lassen sich jedoch auch Domänen realisieren, die in verschiedenen Locales verschiedene Key/Value-Paare besitzen. Die Verwendung einer solchen Domäne muß dann aber mit der übrigen Geschäftslogik genau abgestimmt werden.

Da die Instanz einer Domäne bei entsprechendem Lifecycle in der Session gespeichert werden kann, muß die Domäne serialisierbar sein. Die Domäne darf einen beliebigen Konstruktor haben, der in der *Domain View* mit dem Datenmodell verknüpft wird.

`public interface Domain extends Serializable`

- **`public Iterator getKeys(Locale locale)`:** Liefert einen Iterator über die Keys einer Domäne in der Reihenfolge der angegebenen Locale zurück.
- **`public String getValue(String key, Locale locale)`:** Liefert den locale-spezifischen Value der Domäne für den angegebenen Key zurück.
- **`public boolean containsKey(String key, Locale locale)`:** Prüft, ob der angegebene Key in der durch die Locale bestimmten Key-Menge der Domäne enthalten ist.

6.3.5 DataConversion

Einem Atom wird ein Datentyp zugeordnet. Dies ist eine beliebige Java-Klasse für den Dateninhalt eines Atoms. Die sprachunabhängigen Java-Objekte der Dateninhalte werden von der *zen Engine* aus sprachspezifischen Stringrepräsentationen erstellt bzw. in sprachspezifische Stringrepräsentationen zurückkonvertiert. Für die Konvertierung sind Klassen zuständig, die das Interface *DataConversion* implementieren. Schlägt das Parsen einer Stringrepräsentation in das durch den Datentyp bestimmte Java-Objekt fehl, signalisiert die Konvertierungsklasse durch Werfen einer *ParsingException* einen Benutzerfehler.

`public interface DataConversion`

- **`public Object parse(String string, Locale locale) throws ParsingException`**: Parst den angegebenen String in der angegebenen Locale in ein Java-Objekt. Dieses muß dem Datentyp entsprechen, den diese Konvertierungsklasse definiert. Schlägt das Parsen fehl, wird durch Werfen einer *ParsingException* ein Benutzerfehler ausgelöst.

- **`public String format(Object data, Locale locale)`**: Erzeugt eine sprachspezifische Stringrepräsentation des angegebenen Java-Objekts in der angegebenen Locale. Das Java-Objekt entspricht dem Datentyp, den diese Konvertierungsklasse definiert.

`public class ParsingException extends Exception`

Parameterlose Exception, die einen Fehler beim Parsen einer Stringrepräsentation signalisiert.

6.4 Core-Funktionen

Die *zen Engine* stellt eine Reihe von Funktionen zur Verfügung, die in Geschäftslogik und Stylesheets benutzt werden können. Feldfunktionen sind fest definierte Funktionen, die z.B. interne Bearbeitungszustände der *zen Engine* als Argumente bereitstellen. Zur Verwendung in Stylesheets sind spezielle FOM-Attribute definiert, die teils von der *zen Engine* erzeugt werden, teils auch in Operationen gesetzt werden können.

6.4.1 Feldfunktionen

Feldfunktionen können als Argumente in Operationen, Domänen und Ressourcen verwendet werden. Sie bieten definierte Zugriffe auf interne Bearbeitungszustände der *zen Engine* und weitere nützliche Funktionen:

- *Composition getIOBlock()*: liefert Zugriff auf die nicht-modellierten Anwendungsdaten (nicht als Ressourcen-Argument verwendbar).

- *Boolean hasErrors()*: liefert zurück, ob während der Bearbeitung Fehler aufgetreten sind. Kann sinnvoll nur im Zusammenhang von *erroraware*-Actions verwendet werden.

- *Boolean hasInternalErrors()*: liefert zurück, ob während der Bearbeitung Anwendungsfehler aufgetreten sind. Kann sinnvoll nur im Zusammenhang von *erroraware*-Actions verwendet werden.

- *Boolean hasUserErrors()*: liefert zurück, ob während der Bearbeitung Benutzerfehler aufgetreten sind. Kann sinnvoll nur im Zusammenhang von *erroraware*-Actions verwendet werden.

- *Object getSessionObject(String key)*: liefert das Objekt zurück, das unter dem angegebenen Schlüssel (anzugeben in der Spalte *FF-Argument* des jeweiligen Argumentdialogs) über die Service-API in der Session abgelegt wurde.

- *Date sessionStartDate()*: liefert den Zeitpunkt zurück, in dem die aktuelle Benutzersession begonnen wurde.

- *Date today()*: liefert das aktuelle Datum ohne Uhrzeit zurück.

6.4.2 Spezielle Attribute

Zur Unterstützung der Gestaltung von Stylesheets, insbesondere generischen Stylesheets, aber auch zur Kommunikation zwischen Stylesheets und *zen Engine* sind verschiedene FOM-Attribute mit festgelegter Bedeutung definiert. Diese liegen im Namespace *http://schema.zeos.de/zen/builtin*. Fast alle FOM-Attribute dieser Art werden von der *zen Engine* gesetzt. Wenn im folgenden ausdrücklich beschrieben, können bestimmte Attribute auch in Operations der *zen*-Anwendung definiert werden. Vermieden werden sollte die feste Definition eines *builtin*:-Attributes im Datenmodell, da dies in der Regel zu unerwünschten Ergebnissen führt.

- **builtin:from/builtin:to**: Attribute zur Ein- bzw. Ausgabe eines Listenausschnitts. Wird bei der Eingabe dem Listen-Element das Attribut *builtin:from* hinzugefügt, verbindet die *zen Engine* die Eingabe-Elemente mit den Listen-Elementen aus der Session ab der durch das Attribut definierten Position. Die so angegebene Position darf die Listengröße nicht überschreiten. Bei der Ausgabe einer Liste wird der Listenausschnitt durch die Angabe der Attribute *builtin:from* und *builtin:to* gesteuert, die durch eine Operation gesetzt werden können.

- **builtin:dirty**: Attribut, das nicht ausgegeben, sondern nur in der Session verwendet wird und das Element beim nachfolgenden Request als geändert markiert. In einem Request mit einer *cancel-*, *clear-* oder *nonvalidating*-Action bzw. einem fehlerhaften Request mit einer *erroraware*-Action werden alle Eingabe-Elemente bzw. durch Computation Rules gewonnene Elemente automatisch mit dem Attribut *builtin:dirty* versehen, bevor sie in der Session gespeichert werden. So kann ein nachfolgender fehlerfreier Request eine erneute Datenübernahme veranlassen bzw. auf solchen Elementen basierende Computation Rules auslösen, auch wenn die Elemente nicht geändert wurden. Die Attribute werden dann wieder automatisch gelöscht. In Spezialfällen kann das Attribut auch durch Operationen gesetzt bzw. gelöscht werden, um eine erneute Datenübernahme auszulösen.

- **builtin:error**: die *zen Engine* markiert so Elemente, die Benutzerfehler ausgelöst haben. Dies betrifft Elemente aus dem Request, die in der *Request Validation* als fehlerhaft erkannt werden, und alle Elemente, die als Parameter einer Operation definiert sind und die eine *OperationException* bzw. eine *ValidationRuleException* wirft. Diese Markierung kann vom Stylesheet-Entwickler genutzt werden, um fehlerhafte Elemente in der Ausgabe zu kennzeichnen. Zur benutzerdefinierten Markierung von Elementen innerhalb von Operationen enthält die *OperationException* eine entsprechende Methode.

- **builtin:domain**: die *zen Engine* integriert Domäneninhalte an zentraler Stelle in die Ausgabe. Jedes Atom, dem eine Domäne zugeordnet ist, erhält ein solches Attribut mit dem Domänennamen als Inhalt. So kann im Stylesheet ein Atom mit seinen Domänenwerten verknüpft werden.

- **builtin:type**: bei Aktivierung der Anwendungseigenschaft *Output Options-Include FOM Types* gibt die *zen Engine* in diesem Attribut den FOM-Element-Typ (*atom*, *comp*, *list*) für jedes Element der Ausgabe an. Dieser kann vom Stylesheet-Entwickler genutzt werden, um generische Stylesheets zu entwickeln, die die Ausgabe anhand der FOM-Struktur aufbauen. Wird für das Stylesheet *generic.xsl* benötigt.

- **builtin:datatype**: bei Aktivierung der Anwendungseigenschaft *Output Options-Include Datatypes* gibt die *zen Engine* für jedes Atom der Ausgabe den Datentyp als vollqualifizierten Java-Klassennamen (z.B. *java.lang.Integer*) an. Dieser kann vom Stylesheet-Entwickler genutzt werden, um z.B. in Stylesheets Atomen je nach Datentyp bestimmte HTML-Formular-Elemente zuzuordnen. Wird für das Stylesheet *generic.xsl* benötigt.

- **builtin:readonly**: bei Aktivierung der Anwendungseigenschaft *Output Options-Include Read Only* gibt die *zen Engine* für jedes Atom der Ausgabe dieses Attribut mit dem Wert *true* an, wenn es im auszugebenden State Node als *out*-Element, aber nicht als *in*-Element zugeordnet wurde. Dies kann vom Stylesheet-Entwickler genutzt werden, um z.B. in Stylesheets HTML-Formular-Elemente zu deaktivieren bzw. Atominhalte als Text auszugeben. Werden Atome als *opt-in* markiert, kann das Atom fallweise Teil der Eingabe sein. Für entsprechend aufgebaute Stylesheets kann das Attribut in diesem Fall auch von Operationen gesetzt werden. Dieses Attribut wird generell für das Stylesheet *generic.xsl* benötigt.

- **builtin:length**: bei Aktivierung der Anwendungseigenschaft *Output Options-Include Length* gibt die *zen Engine* für jedes Atom der Ausgabe in diesem Attribut die ggf. im Datenmodell definierte Länge an. So kann im Stylesheet z.B. die Länge von HTML-Eingabe-Elementen beschränkt werden. Dieses Attribut wird im Stylesheet *generic.xsl* genutzt.

- **builtin:mandatory**: bei Aktivierung der Anwendungseigenschaft *Output Options-Include Mandatory* kennzeichnet die *zen Engine* jedes Atom der Ausgabe, das im Datenmodell als *mandatory* definiert ist, mit diesem Attribut und dem Inhalt *true*. So können im Stylesheet z.B. Pflichtfelder besonders hervorgehoben werden. Dieses Attribut wird im Stylesheet *generic.xsl* genutzt

Weitere builtin:-Funktionen werden im Kapitel 6.6.1 Eingabeformate beschrieben

6.5 Hook API

In Operations wird die Geschäftslogik einer *zen*-Anwendung implementiert. Die *zen Engine* stellt darüber hinaus mit der *Hook API* im Archiv *zenapi.jar* (Package *de.zeos.zen.api.hook*) verschiedene Möglichkeiten zur Anbindung von technischem, anwendungsspezifischem Java-Code zur Verfügung. Der Java-Code wird ähnlich wie bei Operations im Repository eingetragen und zur Laufzeit aufgerufen. Im Unterschied dazu unterliegt Hook-Code generell keiner Workflowsteuerung, sondern wird bei jedem Request aufgerufen.

Hier können z.B. spezielle Logsysteme, Security-Lösungen, Personalisierung u.ä. in eine *zen*-Anwendung integriert werden. Die Zuordnung von Hook-Code geschieht per *zen*-Anwendung, Datenformat (defaultmäßig definiert: *xml* z.B. für SOAP, *map* z.B. für HTML-Parameter) und Protokoll (Frontend-Typ, defaultmäßig definiert: *http* für servlet-basierte Frontends, *ejb* für EJB-basierte Frontends).

6.5.1 RequestModificator

Der *RequestModificator* dient zur Veränderung von Request-Daten. Hier können z.B.:

- Eingabedaten von Fremd-Systemen in die von der *zen Engine* benötigten Formate transformiert werden.

- nicht benötigte Daten, die aus technischen Gründen Bestandteil des Requests sind, gefiltert werden.

- zusätzliche Daten zu den modellierten oder nicht modellierten Anwendungsdaten hinzugefügt werden.

Der entsprechende Code implementiert das Interface *RequestModificatorHook* aus dem Archiv *zenapi.jar* und wird im Repository eingetragen.

`public interface RequestModificatorHook`

- **`public void modify(RequestInfo requestInfo):`** Diese Methode wird von der *zen Engine* nach Betreten des Backends und noch vor der Validierung des Requests aufgerufen.

Für das Interface *RequestInfo* existieren verschiedene Erweiterungen, die je nach Datenformat und Protokoll Zugriff auf spezifische Daten ermöglichen. Es ist sicher, bei entsprechender Zuordnung nach Datenformat und Protokoll im Repository das Interface *RequestInfo* auf die passende Erweiterung (z.B. `HttpRequestInfo)` zu casten.

`public interface RequestInfo`

- **`public String getApplicationName():`** Liefert den Namen der Ziel-Applikation des aktuellen Requests zurück. Der *RequestModificator* wurde dieser Applikation im Repository zugeordnet.
- **`public String getDataFormat():`** Liefert das Datenformat des aktuellen Requests zurück (standardmäßig definiert: *xml*, *map*). Der *RequestModificator* wurde diesem Datenformat im Repository zugeordnet.
- **`public String getProtocol():`** Liefert das Protokoll des aktuellen Requests zurück (standardmäßig definiert: *http*, *ejb*). Der *RequestModificator* wurde diesem Protokoll im Repository zugeordnet.
- **`public String getSessionID():`** Liefert die aktuelle Session-ID zurück.
- **`public SessionStatus getSessionStatus():`** Liefert den Status der aktuellen Session zurück. Der Status ist wie folgt definiert:
  - *SessionStatus.UNDEFINED*: der Status der Session hat keinen definierten Zustand
  - *SessionStatus.NEW*: der Request hat eine neue Session erzeugt
  - *SessionStatus.TIMEOUT*: die Session für den aktuellen Request ist abgelaufen
  - *SessionStatus.VALID*: die Session des aktuellen Requests ist gültig
  - *SessionStatus.CLEARED*: die Session des aktuellen Requests kann gelöscht werden
- **`public void setSessionStatus(SessionStatus status):`** der Status der aktuellen Session wird wie angegeben gesetzt

`public interface HttpRequestInfo`

- **`public void setReferer(String referer):`** Setzt den angegebenen String als Referer des Http-Requests.
- **`public String getReferer():`** Liefert den Referer des Http-Requests zurück.
- **`public void setRequestedLocale(Locale locale):`** Setzt die angegebene Locale als Locale des Http-Requests.
- **`public Locale getRequestedLocale():`** Liefert die Locale des Http-Requests zurück.

`public class HttpMapRequestInfo`

- **`public Map getRequestMap():`** Liefert die Parameter-Map des Http-Requests zurück. Die Inhalte können verändert werden.
- **`public void setMapForFOMConstruction(Map map):`** Setzt eine neues Map-Objekt zur Konstruktion des Request-FOM anstelle der Map, die von *getRequestMap* geliefert wird. Dadurch bleibt trotz der Möglichkeit zur Modifikation der Request-Daten die originale Map erhalten. Diese kann dadurch, anders als bei direkter Modifikation, von nachfolgendem Hook-Code über *getRequestMap* ausgewertet werden.

`public class HttpXMLRequestInfo`

- **`public String getRequestString():`** Liefert den XML-String des Http-Requests zurück.
- **`public void setXmlForFOMConstruction(String string):`** Setzt eine neues String-Objekt zur Konstruktion des Request-FOM anstelle des Strings, der von *getRequestString* geliefert wird. Dadurch bleibt trotz der Möglichkeit zur Modifikation der Request-Daten der originale String erhalten. Dieser kann dadurch, anders als bei direkter Modifikation, von nachfolgendem Hook-Code über *getRequestString* ausgewertet werden.

6.5.2 Interceptor

Neben der Möglichkeit, den Request mittels *RequestModificator* zu verändern, kann ein In-Interceptor auch vor der Validierung des Requests durch die *zen Engine* aufgerufen werden. In einem *In-Interceptor* liegen die Request-Daten bereits in Form eines FOM-Baums vor und können unabhängig vom Eingabeformat modifiziert werden. Einer Anwendung können für ein Datenformat und Protokoll beliebig viele In-Interceptoren in einer definierten Reihenfolge zugeordnet werden. Ein In-Interceptor implementiert das Interface *ClientRequestInterceptor*.

Nach der kompletten Bearbeitung eines Requests durch die *zen Engine* kann die erzeugte Response vor dem Zurücksenden an das Frontend durch einen *Out-Interceptor* modifiziert werden. Einer Anwendung können für ein Datenformat und Protokoll beliebig viele Out-Interceptoren in einer definierten Reihenfolge zugeordnet werden. Ein Out-Interceptor implementiert das Interface *ClientResponseInterceptor*.

`public interface ClientRequestInterceptor`

- **`public void processRequest(RequestInfo requestInfo, ResponseInfo responseInfo, Composition data, Composition io, ControlBlockInfo ctrl, RequestRepFunc rrFunc):`** Diese Methode wird von der *zen Engine* nach dem Aufruf eines optionalen *RequestModificators* und noch vor der Validierung des Requests aufgerufen.

Im Vorgriff auf die weitere Bearbeitung können an dieser Stelle über das Interface *ReponseInfo* schon Eigenschaften der Response ausgelesen und verändert werden. Die modellierten Anwendungsdaten liegen als FOM-Baum in der Wurzel *data* vor, die nicht-modellierten Anwendungsdaten können als FOM-Baum über die Wurzel *io* bearbeitet werden. Die Kontrollflußinformationen liegen im Interface *ControlBlockInfo* vor, Repository-Informationen im Interface *RequestReponseFunc*.

`public interface ClientResponseInterceptor`

- **`public void processResponse(RequestInfo requestInfo, ResponseInfo responseInfo, Composition data, Composition io, ControlBlockInfo ctrl, ResponseRepFunc rrFunc, ErrorCollector errorCollector):`** Diese Methode wird von der *zen Engine* nach der Bearbeitung des Requests und vor dem Zurücksenden an das Frontend aufgerufen.

Auch nach der Bearbeitung des Requests durch die Core kann auf die Requestdaten zugegriffen werden. Außerdem können über das Interface *ReponseInfo* Eigenschaften der Response ausgelesen und verändert werden. Die bearbeiteten modellierten Anwendungsdaten liegen als FOM-Baum in der Wurzel *data* vor, auf die bearbeiteten nicht-modellierten Anwendungsdaten kann über die Wurzel *io* zugegriffen werden. Die Kontrollflußinformationen liegen im Interface *ControlBlockInfo* vor, Repository-Informationen im Interface *RequestReponseFunc*. Speziell für Requests mit einer *erroraware*-Action kann auf die gesammelten Anwendungsfehler über das Interface *ErrorCollector* zugegriffen werden.

Das Interface *ResponseInfo* besitzt ähnlich wie *RequestInfo* eine protokollspezifische Subklassenhierarchie und kann im entsprechenden Interceptor auf die passende Subklasse gecastet werden.

`public interface ResponseInfo`

- **`public String getSessionID():`** Liefert die aktuelle Session-ID zurück.
- **`public String getStylesheetName():`** Liefert den Namen des Stylesheets für die Reponse zurück.
- **`public void setStylesheetName(String name):`** Setzt den Namen des Stylesheets für die Response. Damit kann das auf Ebene der Anwendung definierte Stylesheet überschrieben werden. Das Stylesheet sollte nur im StylesheetSelector überschrieben werden, da erst dann alle Response-Daten verfügbar sind.

`public class HttpResponseInfo`

- **`public boolean isCookieable():`** Liefert zurück, ob die Session bzw. der Client Cookies akzeptiert.
- **`public void setCookieable(boolean cookieable):`** Bestimmt, ob für diese Session versucht werden soll, Cookies zu schreiben.
- **`public String getActionURL():`** Liefert die (evtl. durch das Sessionmanagement umgeschriebene) URL zurück, die im auszugebenden HTML-Formular als Formular-Action benutzt werden soll.
- **`public void setActionURL(String actionURL):`** Setzt die (evtl. für das Sessionmanagement umzuschreibende) URL, die im auszugebenden HTML-Formular als Formular-Action benutzt werden soll.

`public class ControlBlockInfo`

- **`public String getState():`** Liefert den Namen des aktuellen State Nodes zurück.
- **`public Iterator getUserErrors():`** Liefert einen Iterator über alle Benutzerfehler zurück, die jeweils über das Interface *ErrorAccessor* gekapselt sind.

`public class ErrorAccessor`

- **`public String getMessage():`** Liefert die Fehlermeldung des Benutzerfehlers zurück.

- **`public String getCausingElementName()`:** Liefert den Namen des verursachenden Elements zurück, sofern dem Benutzerfehler keine Operation zugrunde liegt.

`public class RequestRepFunc`

- **`public StateGate getRequestStateGate()`:** Liefert die Gate-Eigenschaft des im Request enthaltenen State Nodes zurück.
- **`public String getRequestProcessName()`:** Liefert den Namen des Prozesses des im Request enthaltenen State Nodes zurück.

`public class ResponseRepFunc`

- **`public StateGate getRequestStateGate()`:** Liefert die Gate-Eigenschaft des im Request enthaltenen State Nodes zurück.
- **`public String getRequestProcessName()`:** Liefert den Namen des Prozesses des im Request enthaltenen State Nodes zurück.
- **`public StateGate getResponseStateGate()`:** Liefert die Gate-Eigenschaft des State Nodes der Response zurück.
- **`public String getResponseProcessName()`:** Liefert den Namen des Prozesses des State Nodes der Response zurück.

`public final class ErrorCollector`

- **`public boolean isEmpty()`:** Liefert zurück, ob es gesammelte Anwendungsfehler gibt.
- **`public int errorCount()`:** Liefert die Anzahl der gesammelten Anwendungsfehler zurück.
- **`public void add(Exception e)`:** Fügt eine Exception zu den gesammelten Anwendungsfehlern hinzu.
- **`public Iterator getErrorIterator()`:** Liefert einen Iterator über alle gesammelten Anwendungsfehler zurück. Die Menge der Anwendungsfehler kann über den Iterator nicht verändert werden.
- **`public Iterator getRequestValidationErrorIterator()`:** Liefert einen Iterator über alle gesammelten Anwendungsfehler zurück, die unmittelbar während der Validierung des Request entdeckt wurden.
- **`public Iterator getSystemErrorIterator()`:** Liefert einen Iterator über alle gesammelten Anwendungsfehler zurück, die nach der Validierung des Requests (auch als Folgefehler) entdeckt wurden.

6.5.3 StylesheetSelector

Nach Beendigung der Request-Bearbeitung in der *zen Engine* kann unmittelbar vor dem Zurücksenden der Response optional ein Stylesheet ausgewählt und das auf Ebene der Anwendung definierte Stylesheet überschrieben werden.

`public interface StylesheetSelectorHook`

- **`public void select(RequestInfo requestInfo, ResponseInfo responseInfo, Composition data, Composition io, ControlBlockInfo ctrl, ResponseRepFunc rrFunc, ErrorCollector errorCollector)`:** Diese Methode wird von der *zen Engine* nach dem optionalen Aufruf von Out-Interceptoren unmittelbar vor dem Zurücksenden an das Frontend aufgerufen.

Das Stylesheet wird über das Interface *ResponseInfo* gesetzt. Die übrigen Parameter können zur Auswahl des Stylesheets herangezogen werden.

6.6 Schnittstellen

Das Format der Eingabe-Schnittstelle der *zen Engine* hängt vom verwendeten Frontend ab, das Format der Ausgabe-Schnittstelle ist fest definiert. Die Kenntnis der Schnittstellen-Formate ist hilfreich, aber bei Verwendung der Standard-Frontends und des mitgelieferten generischen Stylesheets *generic.xsl* nicht unbedingt nötig, da diese die Konvertierung von Eingabe- und Ausgabe automatisch vornehmen. Werden eigene Frontends entwickelt oder wird ein Stylesheet von Grund auf neu geschrieben, müssen die Formate jedoch eingehalten werden, um die *zen Engine* korrekt zu bedienen.

6.6.1 Eingabeformate

Ein Request an die *zen Engine* enthält Kontrollflußinformationen, modellierte Anwendungsdaten und nicht-modellierte Anwendungsdaten (im weiteren: IO-Daten). Diese werden im Format der Frontend-Backend-Schnittstelle vereinigt, einer hierarchischen Repräsentation, die von der *zen Engine* in einen FOM-Baum konvertiert werden kann. Derzeit werden zwei unterschiedliche Repräsentationen unterstützt:

- **XML**: Ein XML-basiertes Frontend (z.B. das SOAP-Frontend) sendet die Requestdaten direkt in der XML-Repräsentation des FOM-Baumes ans Backend. Diese muß immer vollständig bis hinunter zur Ebene der Atome gestaltet sein, d.h. ein Request darf keinen Teilbaum mit einer Composition oder List abschließen.

- **Map**: Ein Map-basiertes Frontend (z.B. das HTML-Frontend) arbeitet mit Key/Value-Paaren. Die Eingabedaten liegen hier z.B. als Servlet-Parameter vor. Für jedes Atom des Requests entsteht ein Map-Eintrag, dessen Key dem FOM-Pfadausdruck des Atoms entspricht und dessen Value den Dateninhalt des Atoms enthält.

Die einzelnen Teilbäume aus Kontrollflußinformationen, modellierten Daten und IO-Daten werden unter dem Element *root* zusammengefaßt und sind grundsätzlich optional. In XML hat ein Request also grundsätzlich folgende Struktur:

```
<root>
      <ctrl>...</ctrl>    <!-- Teilbaum für Kontrollflußinformation -->
      <data>...</data>    <!-- Teilbaum für modellierte Daten -->
      <io>...</io>        <!-- Teilbaum für IO-Daten -->
</root>
```

Kontrollflußinformationen

Die Kontrollflußinformationen werden im Teilbaum *ctrl* mitgegeben. Zur Workflowsteuerung werden im allgemeinen mindestens die Angabe von State Node und Action benötigt. Optional kann auch die Locale des Requests übermittelt werden. Unterhalb des Teilbaums *ctrl* werden dazu folgende Elemente genutzt:

- **state**: enthält den Namen des aktuellen State Nodes
- **action**: unterhalb des Action-Elements wird ein leeres Element mit dem Namen der auszuführenden Action erwartet
- **locale/country**: enthält den ISO2-Code für die Ländereinstellung
- **locale/language**: enthält den ISO2-Code für die Spracheinstellung

Wird keine Locale angegeben, wird die Default-Runtime Locale der Anwendung benutzt.

Für den State Node formular *und die Action* weiter *sowie die Ländereinstellung* DE *und die Spracheinstellung* de *entsteht dann folgende XML-Eingabe:*

```
<root>
    <ctrl>
        <state>formular</state>
        <action>
            <weiter/>
        </action>
        <locale>
            <country>DE</country>
            <language>de</language>
        </locale>
    </ctrl>
    <data>...</data>
</root>
```

Die Map-Eingabe für den gleichen Request sieht so aus:

```
/root/ctrl/state = formular
/root/ctrl/action/weiter =
/root/ctrl/locale/country = DE
/root/ctrl/locale/language = de
/root/data/...
```

Modellierte Anwendungsdaten

Die Repräsentation der modellierten Anwendungsdaten ergibt sich direkt aus dem Datenmodell einer *zen*-Anwendung. Das Format der Eingabedaten muß der Locale entsprechen, die in den Kontrollflußinformationen angegeben ist, sonst kann die Eingabe zu einem Benutzerfehler führen.

Die XML-Eingabe für eine Wertpapierorder aus der Beispielanwendung könnte folgendermaßen aussehen:

```
<root>
    <ctrl>...</ctrl>
    <data>
        <order>
            <ordertyp>k</ordertyp>
            <wkn>123456</wkn>
            <stueck>1000</stueck>
            <limit>20,80</limit>
            <gueltig-bis>01.01.2004</gueltig-bis>
        </order>
    </data>
</root>
```

Die Map-Eingabe für den gleichen Request sieht so aus:

```
/root/ctrl/...
/root/data/order/ordertyp = k
/root/data/order/wkn = 123456
/root/data/order/stueck = 1000
/root/data/order/limit = 20,80
/root/data/order/gueltig-bis = 01.01.2004
```

Bei XML-Eingabe definiert die Reihenfolge von Listen-Kind-Elementen automatisch deren Reihenfolge in der Liste. Bei Map-Eingaben werden die Listen bzw. Listen-Kind-Elemente mit dem entsprechenden FOM-Pfadausdruck gekennzeichnet.

Der Request zum Streichen der zweiten von zwei laufenden Orders könnte z.B. so aussehen:

```
<root>
    <ctrl>...</ctrl>
    <data>
        <lfd-orders>
            <lfd-order>
                <wkn>123456</wkn>
                <status>ok</status>
            </lfd-order>
            <lfd-order>
                <wkn>987654</wkn>
                <status>gestrichen</status>
            </lfd-order>
        </lfd-orders>
    </data>
</root>
```

Die Map-Eingabe für den gleichen Request sieht so aus:

```
/root/ctrl/....
/root/data/$lfd-orders/lfd-order[0]/wkn = 123456
/root/data/$lfd-orders/lfd-order[0]/status = ok
/root/data/$lfd-orders/lfd-order[1]/wkn = 987654
/root/data/$lfd-orders/lfd-order[1]/status = gestrichen
```

IO-Daten

IO-Daten können genauso wie modellierte Anwendungsdaten an die *zen Engine* gesendet werden. Da IO-Daten nicht modelliert sind, damit auch nicht validiert werden und nur als optionale Möglichkeit zur Datenspeicherung zu verwenden sind, gibt es keine Einschränkung bei der Anwendung, solange die Eingabestruktur syntaktisch korrekt ist.

XML-Request: XML-Deklaration

Werden XML-Daten an die *zen Engine* geschickt, die außer ASCII-Daten z.B. Umlaute oder Sonderzeichen enthalten, sollte in der XML-Deklaration ein passender Zeichensatz angegeben werden, z.B.:

- <?xml version="1.0" encoding="UTF-8"?>
- <?xml version="1.0" encoding="ISO-8859-1"?>

Attribute

Elemente können in der Eingabe mit Attributen versehen werden. In der XML-Eingabe wird dazu das jeweilige Element mit dem gewünschten XML-Attribut versehen. In der Map-Eingabe können die entsprechenden FOM-Pfadausdrücke verwendet werden.

Beispiel zur Verwendung von selbstdefinierten Attributen in der Eingabe:

```
<root>
    <ctrl>...</ctrl>
    <data>
        <order>
            ...
            <limit js-checked="false">20,80</limit>
            ...
        </order>
    </data>
</root>
```

Die Map-Eingabe für den gleichen Request sieht so aus:

```
...
/root/data/order/limit[@js-checked="false"] = 20,80
...
```

***builtin:*-Funktionen**

Werden in einem XML-Request *builtin*:-Attribute verwendet, muß im *root*-Element der *builtin*-Namespace definiert werden:

```
<root xmlns:builtin="http://schema.zeos.de/zen/builtin">...</root>
```

In XML-basierten Frontends existiert neben den allgemein verwendbaren *builtin*:-Attributen eine zusätzliche Verwendungsmöglichkeit des Attributs *builtin:type* zur Kennzeichnung von Listen. Während mehrelementige Listen automatisch erkannt werden können, sind einelementige Listen zunächst nicht von einer Composition zu unterscheiden. Liegt für eine XML-Eingabe gleichzeitig kein Schema vor, das die Liste eindeutig definiert, kann die Liste durch das Attribut *builtin:type*="*list*" gekennzeichnet werden.

Für HTML-basierte Frontends stehen zwei *builtin*:-Funktionen für die Auswertung von Checkboxen und Radiobuttons zur Verfügung. Eine auf einer HTML-Seite nicht angehakte Checkbox bzw. eine Radiobutton-Gruppe, in der kein Radiobutton selektiert wird, wird vom Browser nicht an die *zen Engine* gesendet. Wenn die betreffenden Elemente im Datenmodell nicht als *opt-in* gekennzeichnet sind, verstößt der Request daher gegen das Datenmodell und

löst damit einen Anwendungsfehler aus. Sollen diese Elemente jedoch bewußt nicht als *opt-in* gekennzeichnet werden, können auf einer HTML-Seite standardmäßig für jede Checkbox bzw. für jede Radiobutton-Gruppe zusätzliche *HIDDEN*-Fields mitgeschickt werden, die die Existenz der entsprechenden Elemente in jedem Fall signalisieren. In den *HIDDEN*-Fields wird der Name (FOM-Pfad) der Checkbox bzw. der Radiobutton-Gruppe um einen Markierungsstring erweitert:

- `<input type="hidden" name="{$name}.builtin:check" value=""/>`
- `<input type="hidden" name="{$name}.builtin:radio" value=""/>`

Diese Vorgehensweise wird z.B. vom Komponenten-Stylesheet *components.xsl*, das auch von *generic.xsl* verwendet wird, genutzt.

6.6.2 Ausgabeformat

Die von der *zen Engine* erzeugte Ausgabe ist immer ein XML-String. Dieser enthält Kontrollflußinformationen, modellierte Anwendungsdaten, IO-Daten und Domänen-Daten. Die ausgegebenen XML-Knoten sind teilweise durch *builtin*:-Attribute und Ressourcen ergänzt: Element-Ressourcen werden an den modellierten Elementen ausgegeben, Action-Ressourcen an den Action-Elementen unterhalb des *ctrl*-Teilbaums und State-Ressourcen am *root*-Element.

Die Ausgabe startet mit der XML-Deklaration und dem *root*-Element, an dem alle verwendeten Namespaces (*builtin* für *builtin*:-Attribute und *resource* für Ressourcen), die modellierten State-Ressourcen und State-Attribute angebracht sind.

Im folgenden wird die Ausgabe am Beispiel des Börsenorderformulars erklärt. Am State Node formular *wurde für dieses Beispiel eine Ressource mit dem Namen* headline *und dem Inhalt „Ordereingabe“ modelliert, die hier in der Ausgabe als State-Ressource ins root-Element geschrieben wird.*

```
<?xml version="1.0" encoding="UTF-8"?>
<root xmlns:resource="http://schema.zeos.de/zen/resource"
    xmlns:builtin="http://schema.zeos.de/zen/builtin"
    resource:headline="Ordereingabe">
...
</root>
```

Kontrollflußinformationen

Die Kontrollflußinformationen enthalten den ausgegebenen State Node sowie alle modellierten Actions, die von diesem State Node ausgehen. Zu jeder Action werden auch die modellierten Ressourcen und Attribute ausgegeben.

Eine fehlerfreie Response enthält z.B. folgende Kontrollflußinformationen:

```
<root ...>
    <ctrl>
        <locale>
            <country>DE</country>
            <language>de</language>
        </locale>
        <actions>
            <action>
                <weiter resource:src="/buttons/de/order.gif"/>
            </action>
        </actions>
        <state>formular</state>
    </ctrl>
</root>
```

Da der State Node formular *nur eine Action* (weiter) *enthält, ist die Liste der ausgegebenen Actions einelementig. Für Action wurde auch eine Action-Ressource* src *modelliert, die mit ausgegeben wird.*

Enthält der auszugebende State Node Benutzerfehler, werden diese zentral im *ctrl*-Teilbaum mit ausgegeben, zusätzlich noch als *builtin:error* bei denjenigen modellierten Elementen, die zum Benutzerfehler geführt haben (siehe weiter unten).

Zentrale Ausgabe von Benutzerfehlern:

```
<root ...>
    <ctrl>
        <locale>...</locale>
        <actions>...</actions>
        <state>...</state>
        <errors>
            <error>Bitte geben Sie für eine Order eine Stückzahl an</error>
            <error>Bitte geben Sie das Datum in der Form 01.01.2004
an</error>
        </errors>
    </ctrl>
</root>
```

Modellierte Anwendungsdaten

Die modellierten Anwendungsdaten werden mit allen ihren Attributen und Ressourcen ausgegeben. Da das Element *data* in jeder Response ausgegeben wird, werden dort modellierte Attribute und Ressourcen automatisch in jedem State Node ausgegeben. Existieren bereits Dateninhalte zu einem Atom in der Session, so werden diese mit ausgegeben. Gleiches gilt, wenn der Workflow aufgrund eines Benutzerfehlers nicht weitergeschaltet wurde. Dann enthält die Ausgabe jedes Atoms die jeweils zuvor eingegebenen Daten. Jedes Element kann ggf. noch mit weiteren *builtin*:-Attributen versehen werden (s. 6.4.2 Spezielle Attribute).

Die Ausgabe des noch leeren State Nodes formular *mit Beispiel-Ressourcen* intro*,* label*, und* popup-link:

```
<root ...>
    <ctrl>...</ctrl>
    <data>
        <order resource:intro="Bitte geben Sie hier ihre Orderdaten ein">
            <ordertyp resource:label="Kauf/Verkauf"
                builtin:domain="order-types"></ordertyp>
            <wkn resource:label="WKN" builtin:length="6"></wkn>
            <stueck resource:label="Stück" builtin:length="6"></stueck>
            <limit resource:label="Limit" builtin:length="8"></limit>
            <gueltig-bis resource:label="Gültig bis"
                resource:popup-link="Kalender zeigen"
                builtin:length="10"></gueltig-bis>
        </order>
    </data>
    <domains>...</domains>
</root>
```

War ein Element an einem Benutzerfehler beteiligt, wird dieser Fehler, neben der Ausgabe unter *ctrl/errors*, auch noch dem Element als Attribut *builtin:error* hinzugefügt.

Die Ausgabe des teilweise fehlerhaft gefüllten State Nodes formular:

```
<root ...>
    <ctrl>...</ctrl>
    <data>
        <order resource:intro="Bitte geben Sie hier ihre Orderdaten ein">
            <ordertyp resource:label="Kauf/Verkauf"
                builtin:domain="order-types">k</ordertyp>
            <wkn resource:label="WKN" builtin:length="6"
                builtin:error="Bitte geben Sie eine gültige WKN an">abc</wkn>
            <stueck resource:label="Stück" builtin:length="6">100</stueck>
            <limit resource:label="Limit" builtin:length="8"
                builtin:error="Eingabe unvollständig"></limit>
            <gueltig-bis resource:label="Gültig bis"
                resource:popup-link="Kalender zeigen"
                builtin:length="10">1.1.04</gueltig-bis>
        </order>
    </data>
    <domains>...</domains>
</root>
```

IO-Daten

Da für IO-Daten kein Modell vorhanden ist, können sie auch nicht mit Ressourcen verknüpft werden. Es werden immer alle verfügbaren IO-Daten zusammen ausgegeben, da sie, anders als normale Daten, keiner In/Out-Zuweisung unterliegen.

Bei einer Http-Response wird im IO-Datenbereich zusätzlich das Element *builtin:target* hinzugefügt, in das die URL der Anwendung inklusive der eventuell notwendigen Session-ID eingetragen ist. Die URL kann in HTML-Formularen dazu genutzt werden, die FORM-Actions korrekt zusammenzusetzen.

Die Response mit einem builtin:target-Element kann z.B. so aussehen:

```
<root ...>
    <ctrl>...</ctrl>
    <data>...</data>
    <io>

<builtin:target>/zen/test;jsessionid=re27wues12x200f7</builtin:target>
    </io>
</root>
```

Domänen

Nur wenn ein Out-Atom eine Domäne nutzt, wird diese mit ausgegeben. Die Ausgabe einer Domäne erfolgt in einer vorgegebenen Struktur immer zentral unter dem Element root/*domains*. Dieser FOM-Teilbaum kann nicht für Eingaben verwendet werden. Im jeweiligen Atom wird die Domäne per Namen referenziert, der im Atom vorhandene Dateninhalt definiert nicht den Domänenwert, sondern den ausgewählten Domänenschlüssel. Durch diese Vorgehensweise müssen Domänen nicht mehrfach ausgegeben werden, wenn sie in verschiedenen ausgegebenen Atomen vorkommen. Im Komponenten-Stylesheet *components.xsl* sind gängige Visualisierungen von Domänen, z.B. als Selectbox oder Radiobuttons, realisiert.

Beispielhafte Ausgabe einer Domäne:
```
<root ...>
    <ctrl>...</ctrl>
    <data>
        <order>
            <ordertyp builtin:domain="order-types">k</ordertyp>
        </order>
    </data>
    <domains>
        <order-types>
            <entry>
                <key>k</key>
                <value>Kauf</value>
            </entry>
            <entry>
                <key>v</key>
                <value>Verkauf</value>
            </entry>
        </order-types>
    </domains>
</root>
```

Fehlerzustand

Schwerwiegende Fehler, die nicht im Rahmen des Workflows abgearbeitet werden können, führen zu einer fest definierten Fehlerausgabe, die vom Frontend erzeugt wird. Diese muß im im XSL-Stylesheet der Anwendung berücksichtigt werden.

Fehlerzustand:
```
<root xmlns:builtin="http://schema.zeos.de/zen/builtin">
    <ctrl>
        <state>builtin:fatal</state>
    </ctrl>
</root>
```

6.7 Frontends

Die *zen Engine* enthält mit der Klasse *GenericServlet* ein Servlet-Frontend für normale Einsatzzwecke. Dieses kann XML-Requests und Map-Requests bearbeiten. Für Spezialfälle kann ein eigenes Servlet durch Vererbung der Klasse *StatefulBaseServlet* bzw. *StatelessBaseServlet* erstellt werden. Bei EJB-Frontends muß grundsätzlich eine Klasse erstellt werden, die von der Klasse *BaseKernelClient* abgeleitet werden kann.

6.7.1 GenericServlet

Das *GenericServlet* ist für normale Einsatzzwecke ausreichend. Es ist abgeleitet vom *StatefulBaseServlet* und integriert daher eine automatische Sessionverwaltung. Der Start einer Anwendung über die Klasse *de.zeos.zen.tomcat.TomcatStart* setzt auf das *GenericServlet* auf. Es nutzt folgende Standardeinstellungen:

- Der Servletkontext ist standardmäßig */zen*.
- Der gewünschte Anwendungsname kann in der URL angegeben werden. So ergibt sich mit den Standardeinstellungen beispielsweise für die Anwendung *order* die URL */zen/order*. Wird keine Anwendung angegeben, nutzt das *GenericServlet* den Anwendungsnamen *default.*
- Der gewünschte Repository-Name kann in der URL oder in dem Property-File *zen.properties* unter dem Schlüssel *de.zeos.zen.ext.frontend.generic.repository* angegeben werden. Wird das Repository per URL ausgewählt, ergibt sich z.B. für die Anwendung *order* und das Repository *order.repository* die URL /zen/order/order.repository. Wird kein Repository-Name angegeben, nutzt das *GenericServlet* als Repository-Namen *zen.repository*.
- Das *GenericServlet* erwartet, daß das Backend unter dem Namen *Kernel* in der Deployment-Konfiguration definiert ist.
- Schlägt die Stylesheet-Auswahl nach kritischen Fehlerfällen fehl, nutzt das *GenericServlet* als Fallback das Stylesheet *generic.xsl* aus dem Java-Archiv *zenapi.jar.*
- Als *ResourceRepository* für Stylesheets wird standardmäßig *<anwendungsname>.xsl* verwendet. Ist dieses *ResourceRepository* nicht in der Datei *scfservice.xml* definiert, wird als Fallback auf das Stylesheet *generic.xsl* aus dem Java-Archiv zenapi.jar zurückgegriffen.
- Schlägt die Ausführung der XSL-Komponente fehl, wird die Seite /error/500.html aufgerufen.

6.7.2 StatefulBaseServlet, StatelessBaseServlet

Reicht das *GenericServlet* nicht als Frontend aus, kann ein eigenes Frontend von *StatefulBaseServlet* (falls automatisches Sessionmanagement benötigt wird) bzw. von *StatelessBaseServlet* (ohne Sessionmanagement) abgeleitet werden. Dabei müssen mindestens folgende Methoden implementiert werden:

- **`public String getApplicationName(Object context)`**: Liefert den Namen der Anwendung zurück, die über dieses Servlet aufgerufen werden soll bzw. für die dieses Servlet das Frontend ist. Das Objekt *context* ist bei Servlets immer der *HttpServletRequest*.
- **`public boolean usePrivateFrontend()`**: Liefert zurück, ob das Frontend ein individuelles Sessionmanagement nutzen möchte.
- **`public String getKernelName()`**: Liefert den Namen der Komponenten zurück, die das Backends abbildet. Die Komponente muß in der Deployment-Konfiguration definiert sein. Momentan ist als Backend-Komponente nur
  de.zeos.zen.core.comp.Kernel
  erlaubt.
- **`public String getRepositoryName(Object context)`**: Liefert den zu verwendenden Repositorynamen zurück. Das Objekt *context* ist bei Servlets immer der *HttpServletRequest.*
- **`public String getFallbackStylesheetName()`**: Liefert zurück, welches Stylesheet das Frontend benutzen soll, wenn die Stylesheet-Auswahl in kritischen Fehlerfällen fehlschlägt. Die Rückgabe kann entweder ein Stylesheet im Klassenpfad beschreiben, oder inklusive des *url*- oder *file*-Protokolls angegeben werden.
- **`public String getFallbackPage()`**: Liefert zurück, auf welche statische URL das Frontend umleiten soll, wenn die XSL-Komponente zur Ausgabe fehlschlägt.
- **`public String getStylesheetResourceRepositoryName(Object context)`**: Liefert den Namen des *ResourceRepository*, in dem die Ausgabe-Stylesheets abgelegt sind, oder *null*, wenn kein eigenes *ResourceRepository* existiert oder das im Java-Archiv *zenapi.jar* mitgelieferte generische Stylesheet verwendet werden soll. Das Objekt *context* ist bei Servlets immer der *HttpServletRequest.*

Zusätzlich können folgende Methoden überschrieben werden, um die Request/Response-Verarbeitung zu verändern:

- **`protected String modifyInputStream(HttpServletRequest request, String data)`**: wird bei Requests mit den Mime-Types *text/xml* und *application/soap* aufgerufen. In *request* ist der originale Http-Request enthalten, in *data* der Requestinhalt als XML-String. Dieser kann modifiziert und zurückgegeben werden.
- **`protected HashMap modifyKeyValuePairs(HttpServletRequest request, HashMap map)`**: wird bei sonstigen Requests aufgerufen. In *request* ist der originale Http-Request enthalten, in *map* aufbereitete Key/Value-Paare des Requests. Diese können bearbeitet und zurückgegeben werden.

6.7.3 BaseKernelClient

Um ein EJB-Frontend zu erstellen, wird eine neue Klasse von *BaseKernelClient* abgeleitet. Hier müssen die gleichen Methoden implementiert werden, wie bei einer Erweiterung des Stateful- bzw. StatelessBaseServlets.
Die Klasse *BaseKernelClient* stellt jeweils einen Konstruktor für Map- und XML-Eingaben zur Verfügung. Davon muß mindestens einer überschrieben werden.

Beispiel für ein EJB-Frontend:

```
public class MyKernelClient extends BaseKernelClient {
  public MyKernelClient(Map map) {
    super(map);
  }
  public MyKernelClient(String xml) {
    super(xml);
  }
  public String getApplicationName(Object context) {
    return "myapp";
  }
  (...)
  public String getFallbackPage() {
    return null;
  }
}
```

Zur Anwendung eines EJB-Frontends stellt die Klasse *BaseKernelClient* zusätzliche Methoden zur Verfügung:

- **`public void callBackend()`**: Ruft das Backend auf.
- **`public EJBResponseInfo getResponseInfo()`**: Liefert Informationen zur Response zurück.
- **`public String getXML()`**: Liefert die XML-Rückgabe des Backends zurück.

- **`public Object getTransformedResult(String repositoryName, String xslFile)`**: Transformiert die XML-Rückgabe des Backends mit dem Stylesheet *xslFile.* Das Stylesheet wird relativ über das *ResourceRepository* mit dem Namen *repositoryName* bezogen. Das Ergebnis der Transformation wird als *Object* zurückgegeben, da die Klasse von der XSL-Transformationsmethode abhängt. Bei einer Transformation in HTML, XML, WML oder Text ist das Ergebnis vom Typ *String*, bei einer Transformation in PDF vom Typ *byte[].*

Beispiel für die Nutzung eines EJB-Frontends:

```
...
MyKernelClient client = new MyKernelClient(requestXML);
client.callBackend();
Object transformedResult = client.getTransformedResult(repositoryName,
xslFile);
```

6.8 XSL-Stylesheets

Die XML-Rückgabe eines Backend-Aufrufs wird mit den Standard-Frontends automatisch mittels eines XSL-Prozessors in ein Zielformat transformiert. Für die Entwicklung stehen im Archiv *zenapi.jar* die generischen Stylesheets *generic.xsl* für HTML-Seiten und *xml.xsl* für XML-Ausgaben zur Verfügung.

Der Name des jeweils zu verwendenden Stylesheets wird entweder fest in der *Application View* konfiguriert oder zur Laufzeit durch einen anwendungsspezifisch modellierten *StylesheetSelector* vorgegeben. Der Pfad, aus dem dieses Stylesheet gelesen wird, ergibt folgendermaßen:

- **GenericServlet**: Das *GenericServlet* sucht in der Service-Konfiguration nach einem *ResourceRepository* mit Namen *<anwendungsname>.xsl*. Existiert es, wird das Stylesheet aus diesem gelesen. Beispiel: Anwendungsname ist *myApp*, dann wird im *ResourceRepository myApp.xsl* nachgeschlagen.

- **Selbsterstelltes Frontend**: Wird ein eigenes Frontend von StatefulBaseServlet bzw. von StatelessBaseServlet abgeleitet, kann mittels Überschreiben der Methode
  `public String getStylesheetResourceRepositoryName(Object context)`
  ein *ResourceRepository* vorgegeben werden (s. Kapitel 6.7.2 StatefulBaseServlet, StatelessBaseServlet).

- **Fallback**: Falls auf die oben beschriebene Weise kein gültiges *ResourceRepository* ermittelt werden kann, wird automatisch das mitgelieferte generische Stylesheet aus dem Java-Archiv *zenapi.jar* verwendet.

6.8.1 generic.xsl

Dieses Stylesheet aus dem Java-Archiv *zenapi.jar* stellt auf generische Weise beliebige XML-Rückgaben des Backends in einer einfachen HTML-Visualisierung dar. Damit läßt sich eine Web-Anwendung entwickeln bzw. ein funktionaler Prototyp einer Web-Anwendung erstellen, ohne sich mit der Oberflächen-Entwicklung auseinandersetzen zu müssen. Es wird in der Standard-Konfiguration für jede neue *zen*-Anwendung verwendet.

Für das Stylesheet müssen in der *Application View* die Anwendungseigenschaften *Include FOM Types, Include Datatypes* und *Include Read Only* gesetzt sein. Es importiert zusätzlich das Komponenten-Stylesheet *components.xsl*.

Domäneninhalte werden grundsätzlich als Selectboxen dargestellt, Radiobuttons werden in diesem Stylesheet nicht verwendet. Um eine Selectbox mit multipler Auswahl darzustellen, muß im Datenmodell eine Liste von Atomen definiert werden. Dem Atom wird eine Domäne mit der auszuwählenden Wertemenge zugewiesen. Dieses Stylesheet erwartet, daß das Atom zusätzlich mit dem Attribut *multi* versehen wird.

Im folgenden werden die einzelnen Templates des Stylesheets beschrieben.

Template match="/"

Dieses Template definiert den Seitenrahmen. Es unterstützt einige wenige Ressourcen zur Seitengestaltung, so z.B. die State-Ressource *headline* für eine Überschrift oder die *data*-Element-Ressource *error*, die etwaigen Benutzerfehlern vorangestellt wird. Die nötigen Kontrollflußinformationen werden als *hidden fields* definiert. Ansonsten werden Actions bzw. der *data*-Teilbaum in weiteren Templates bearbeitet.

Template name="actions"

Hier werden die Actions dargestellt. Wurden Action-Ressourcen namens *src* vergeben, wird der Ressourcen-Inhalt als Bildpfad interpretiert. Die Actions werden dann als Image-Buttons dargestellt. Wurden stattdessen Action-Ressourcen namens *label* vergeben, erfolgt die Darstellung als einfache Submit-Buttons mit dem Ressourcen-Inhalt als Beschriftung. Ohne Angabe von Ressourcen wird der Action-Name als Beschriftung der Submit-Buttons verwendet. Die Actions werden nach Inhalt der *label*- bzw. *src*-Ressource sortiert.

Template name="element"

Dieses Template delegiert die weitere Bearbeitung von Elementen anhand ihres FOM-Typs an typspezifische Templates. Eine Liste von Atomen, die mit dem Attribut *multi* gekennzeichnet sind, wird als Spezialfall direkt im Atom-Template als Multi-Selectbox ausgegeben.

Template name="list"

Hier wird eine Liste dargestellt. Besteht die Liste aus Atomen, die nicht mit dem Attribut *multi* gekennzeichnet sind, wird die *label*-Ressource der Atom-Definition als Überschrift einer einspaltigen Tabelle verwendet. Für jeden Listeneintrag wird das Atom-Template in einer neuen Tabellenzeile aufgerufen. Bei einer Liste aus Compositions werden die *label*-Ressourcen der Kindelemente der Composition als Überschrift einer mehrspaltigen Tabelle verwendet. Jedes Kindelement der Composition definiert eine Spalte. Für jeden Listeneintrag wird eine neue Tabellenzeile erzeugt. Für jedes Kindelement der Composition wird der entsprechende Spalteninhalt über den Aufruf des Element-Templates erzeugt. So kann z.B. eine Tabelle aus Eingabefeldern erzeugt werden.

Template name="comp"

Wenn eine Composition nicht – wie im vorigen Template beschrieben – direkt unter einer Liste definiert ist, werden die Kindelemente untereinander dargestellt. Jedes Kindelement wird mit dem Inhalt seiner *label*-Ressource versehen. Die weitere Bearbeitung der Kindelemente übernimmt wieder das Element-Template.

Template name="atom"

In diesem Template werden Atome als Eingabe-Elemente dargestellt. Atome mit zugeordneten Domänen werden als Selectboxen dargestellt, Atome mit Boolean-Datentypen als Checkboxen. Atome mit dem Attribut *password* werden als Passwort-Eingabefelder dargestellt, alle anderen Atome werden als Text-Eingabefelder ausgegeben. Die Darstellung erfolgt über Templates, die vom importierten Stylesheet *components.xsl* zur Verfügung gestellt werden.

6.8.2 components.xsl

Dieses Stylesheet enthält generische Visualisierungen von HTML-Eingabe-Elementen. Es wird vom Stylesheet *generic.xsl* verwendet, kann aber auch in eigene HTML-basierte Stylesheets importiert werden. Für die korrekte Darstellung wird die Anwendungseigenschaft *include FOM Types* benötigt.

Template name="path"

Internes Template, das die FOM-Pfadausdrücke für Elemente konstruiert

Template name="input-checkbox"

Template für die Darstellung einer Checkbox. Folgende Parameter sind definiert:

- **element**: Atom, das als Checkbox dargestellt werden soll.
- **style**: Erzeugt innerhalb des *input*-Tags ein *style*-Attribut mit dem angegebenen Inhalt.
- **readonly**: Die Checkbox wird als Text oder Bild dargestellt. Letzteres erfordert die zusätzliche Angabe des Parameters *readonly-img-path.*
- **readonly-img-path**: Verzeichnis-Pfad für die readonly-Darstellung. Die Namen der Bilddateien sind als *check_readonly_on.gif* bzw. *check_readonly_off.gif* vorgegeben.

Template name="input-text"

Template für die Darstellung eines Text-Eingabefeldes. Das Attribut *maxlength* des *input*-Tags wird automatisch aus dem FOM-Attribut *builtin:length* gefüllt. Folgende Parameter sind definiert:

- **element**: Atom, das als Eingabefeld dargestellt werden soll.
- **size**: Erzeugt innerhalb des *input*-Tags ein *size*-Attribut mit dem angegebenen Inhalt.
- **style**: Erzeugt innerhalb des *input*-Tags ein *style*-Attribut mit dem angegebenen Inhalt.
- **readonly**: Der Atom-Inhalt wird als Text dargestellt.
- **password**: Das *input*-Tag wird als *type="password"* erzeugt.

Template name="input-image"

Template für die Darstellung einer Action als Image-Button. Folgende Parameter sind definiert:

- **element**: Action, die als Image-Button dargestellt werden soll.
- **src**: Bildpfad für den Image-Button. Defaultmäßig wird der Inhalt der FOM-Ressource *src* genutzt.

Template name="input-submit"

Template für die Darstellung einer Action als Standard-Submit-Button. Folgende Parameter sind definiert:

- **element**: Action, die als Image-Button dargestellt werden soll.
- **label**: Text für den Submit-Button. Standardmäßig wird der Inhalt der FOM-Ressource *label* genutzt.

Template name="input-select"

Template für die Darstellung eines Atoms mit zugeordneter Domäne als Selectbox. Die Domänen-Werte werden in der Selectbox als Einträge angezeigt. Folgende Parameter sind definiert:

- **element**: Atom, das als Selectbox dargestellt werden soll.
- **style**: Erzeugt innerhalb des *input*-Tags ein *style*-Attribut mit dem angegebenen Inhalt.
- **degenerate**: Besitzt die Selectbox nur einen Eintrag, wird bei Aktivierung dieser Eigenschaft der Domänen-Wert des Atoms als Text dargestellt.
- **readonly**: Der Domänen-Wert des Atoms wird als Text dargestellt. Bei Multi-Selectboxen werden die Domänen-Werte aller Atome mit *
* getrennt als Text ausgegeben.

Template name="value-select"

Template für die Rückgabe des Domänen-Wertes eines Atoms. Folgende Parameter sind definiert:

- **element**: Atom, dessen Domänen-Wert zurückgeliefert werden soll.

Template name="input-radio"

Template für die Darstellung eines Atoms mit zugeordneter Domäne als Radiobutton-Gruppe. Für jeden Domänen-Eintrag wird ein Radiobutton mit dem Domänen-Wert als Textlabel erzeugt. Folgende Parameter sind definiert:

- **element**: Atom, das als Radiobutton-Gruppe dargestellt werden soll.
- **align**: Gibt an, ob das Textlabel links (*left*) oder rechts (*right*) vom Radiobutton stehen soll.
- **spacer**: Gibt an, ob die Radiobutton-Textlabel-Paare voneinander durch ein * * (Leerzeichen) getrennt werden sollen.
- **break**: Gibt an, ob nach einem Radiobutton-Textlabel-Paar ein Zeilenumbruch *
* erzeugt werden soll.

Template name="input-radio-single"

Template für die Darstellung eines einzigen Domänen-Eintrags eines Atoms als Radiobutton. Mit diesem Template können z.B. manuell Textlabels und zugehörige Radiobuttons getrennt in einzelne Tabellenzellen plaziert werden. Dieses Template wird dazu für jeden möglichen Domänen-Eintrag der Domäne aufgerufen. Folgende Parameter sind definiert:

- **element**: Atom, das als Radiobutton dargestellt werden soll.
- **key**: Domänen-Key, für den der Radiobutton erzeugt werden soll.
- **align**: Gibt an, ob das Textlabel links (*left*) oder rechts (*right*) vom Radiobutton stehen soll.
- **show**: Gibt an, ob der Radiobutton angezeigt werden soll. Ansonsten wird nur das Textlabel angezeigt.
- **spacer**: Gibt an, ob Radiobutton und Textlabel durch ein * * getrennt werden sollen.
- **readonly**: Der Radiobutton wird als Text oder Bild dargestellt. Letzteres erfordert die zusätzliche Angabe des Parameters *readonly-img-path*.
- **readonly-img-path**: Verzeichnis-Pfad für die readonly-Darstellung. Die Namen der Bilddateien sind als *radio_readonly_on.gif* bzw. *radio_readonly_off.gif* vorgegeben.

6.8.3 xml.xsl

Dieses Stylesheet formatiert die XML-Rückgabe des Backends und reicht diese durch.

7 Anwendungsdesign

In diesem Kapitel werden Hinweise zur Modellierung und Implementierungen von *zen*-Anwendungen gegeben. Im ersten Teil werden grundlegende Überlegungen vorgestellt, im zweiten Teil werden komplexere Patterns erläutert, wie sie in realen Anwendungen vorkommen.

7.1 Basis

Die Realisierung einer *zen*-Anwendung umfaßt die Modellierung der Anwendung im *zen Developer*, die Implementierung der Java-Geschäftslogik und die Erstellung eines XSL-Stylesheets.

7.1.1 Technische Beschränkungen

Die Möglichkeiten bei der Implementierung der Geschäftslogik werden von der *zen Platform* nicht eingeschränkt. Sofern man sich allerdings alle Skalierungsoptionen offen halten will, muß man folgendes beachten:

- Technische Dienste müssen immer über die Service-API genutzt werden, auch wenn je nach Ablaufumgebung auch der direkte Aufruf an der *zen Engine* vorbei möglich sein könnte.
- Der Zugriff auf Entity Beans und Stateful Session Beans nach der EJB-Spezifikation ist derzeit nicht abstrahiert. Werden sie genutzt, legt man damit zumindest fest, daß für diese ein J2EE-Container verfügbar sein muß.
- Der Zugriff auf Stateless Session Beans ist generell in allen Deployments erlaubt, sofern die Beans als SCF-Komponente realisiert sind.

Da über die Service-API alle notwendigen Dienste realisiert sind, ist dies in der Praxis keine Einschränkung. Der Zugriff auf Stateful Session Beans ist mit der *zen Platform* unnötig, da das (verteilte) Sessionhandling über die *zen Engine* zur Verfügung gestellt wird. Die Verwendung von JDO als Persistenzschicht ist in den meisten Fällen deutlich effizienter als die Nutzung schwergewichtiger Entity Beans.

7.1.2 Workflowmodellierung

Decisions vs. Validation Rules

Mit Decision Nodes lassen sich Feedback-Meldungen für Benutzer auf eine eigene Seite auslagern. Die Decision Operation überprüft z.B. bestimmte Eingabedaten einer Formular-Seite auf eine Bedingung. Ist diese erfüllt, kann auf die normale Ziel-Seite weitergeschaltet werden. Ist die Bedingung nicht erfüllt, kann eine Feedback-Seite angezeigt werden.

Soll dem Benutzer jedoch die Möglichkeit gegeben werden, von der Feedback-Seite zurück zum Formular zu gelangen, um nach Anpassung seiner Eingabedaten die Decision erneut zu durchlaufen, sollte statt der Konstruktion über einen Decision Node direkt eine Validation Rule genutzt werden, da diese ein unmittelbares Feedback liefert.

Ein Kunde soll über ein Formular Daten zu seiner persönlichen finanziellen Situation eingeben, um eine Sofortzusage über einen Kredit zu erhalten. Über einen Decision Node kann dann z.B. ein State Node mit der Zusage und ggf. weiteren nötigen Eingaben ausgewählt werden. Im Falle einer nicht erteilten Zusage wird stattdessen ein State Node angezeigt, in dem der Kunde gebeten wird, den Kreditwunsch in einem persönlichen Beratungsgespräch zu klären. Mit einer solchen Workflow-Konstruktion wird das Feedback vom Formular entkoppelt.
Ein anderes Beispiel: Für das Login zur einer Bank-Anwendung gibt der Kunde Kontonummer und PIN an. Ist die angegebene PIN falsch, soll der Kunde in der Lage sein, diese auf einfache Weise zu korrigieren. Hier wäre eine Workflow-Konstruktion wie im vorigen Beispiel mit ausgelagerter Feedback-Seite zu umständlich. Stattdessen kann unmittelbares Feedback über eine Validation Rule gegeben werden.

Action Types

Normalerweise reicht es aus, Workflows mit *default*-Actions zu modellieren. Für manche Situationen bieten *default*-Actions jedoch keine ausreichende Lösung.

Mit einer *cancel*-Action kann z.B. ein Formular verlassen werden, ohne daß z.B. Pflichtfelder ausgefüllt werden müßten. Alle Eingaben des Benutzers in dieses Formular werden verworfen, die Sessiondaten werden durch die Eingaben nicht verändert. Die Objekt-Dateninhalte der Eingaben sind im weiteren Verlauf *null*. Workflow-Operationen, die in diesem Kontext ausgeführt werden, müssen dies berücksichtigen.

Mit einer *clear*-Action kann z.B. ein Formular serverseitig gelöscht werden, indem alle im Formular enthaltenen Felder aus der Session entfernt werden. Da keine Validierung stattfindet, können so auch Formulare mit Pflichtfeldern geleert werden. Die Objekt-Dateninhalte der Eingaben sind im weiteren Verlauf *null*. Workflow-Operationen, die in diesem Kontext ausgeführt werden, müssen dies berücksichtigen.

Mit *nonvalidating*-Actions können z.B. Formular-Eingaben ohne Validierung in die Session übernommen werden. Eine solche Action sollte nur dann verwendet werden, wenn im Workflow sichergestellt ist, daß in einem späteren Schritt eine Validierung der Daten erfolgt. Zusätzlich ist darauf zu achten, daß bei einer *nonvalidating*-Action zwar die Benutzereingaben als Stringrepräsentationen in den Atomen enthalten sind, jedoch keine Konvertierung in die eigentlichen Objekt-Dateninhalte vorgenommen wird. Die Objekt-Dateninhalte der Eingaben sind im weiteren Verlauf also *null*. Workflow-Operationen, die in diesem Kontext ausgeführt werden, müssen dies berücksichtigen.

In einem Assistenten sind in mehreren aufeinanderfolgenden Formularen Eingaben zu tätigen. Die Formulare sind mit „weiter“- und „zurück“-Buttons miteinander verbunden. Beim Klick auf „weiter“ sollen die Eingaben normal validiert werden und auf die nächste Formularseite geschaltet werden. Die „weiter“-Action wird deshalb als default *definiert. Beim Klick auf „zurück“ soll der Benutzer zurück zur vorigen Formularseite gelangen. Würde die „zurück“-Action ebenfalls als* default *definiert werden, müßte der Benutzer erst alle Pflicht-Eingaben der aktuellen Formularseite korrekt ausfüllen, um auf die vorige Seite zu wechseln. Bei einer* cancel-*Action würden hingegen alle Eingaben verworfen werden. Bei Verwendung einer* nonvalidating-*Action kann ohne Validierung zurückgewechselt werden. Alle trotzdem schon getätigten, auch fehlerhaften Eingaben, werden als Strings in die Session übernommen, aber nicht konvertiert. Wird das so ausgefüllte Formular später erneut betreten, sind die Eingaben wieder in den Formular-Feldern vorhanden. Wird dann auf „weiter“ geklickt, werden die Eingaben wie gewohnt validiert und müssen ggf. vom Benutzer korrigiert werden. So ist gesichert, daß letztlich nur korrekte Eingaben in die Objekt-Dateninhalte der Atome gelangen.*

Bei Verwendung von *erroraware*-Actions wird die Bearbeitung eines Requests auch bei Fehlern nicht abgebrochen. Insbesondere werden auch Operationen weiter ausgeführt, die dann auf fehlerhaften oder evtl. auf unvollständigen Daten arbeiten. Dies ist bei der Implementierung von Operationen, die in diesem Zusammenhang genutzt werden, zu berücksichtigen.

Eine erroraware-*Action kann z.B. genutzt werden, um eine bestehende Anwendung mit einer zusätzlichen (Daten-)Schnittstelle zu versehen, die von einer Vielzahl von externen Partnern gefüllt wird. Manche Partner sind evtl. nur in der Lage, die Schnittstelle partiell oder mit unerwartet formatierten Werten zu füllen. Bei einer* default-*Action würde eine fehlerhafte Anfrage nicht angenommen werden bzw. den Workflow nicht weiterschalten. Bei Verwendung einer* erroraware-*Action kann der Workflow grundsätzlich weitergeschaltet werden. Ein nachgeschalteter Decision Node kann entscheiden, ob die Anfrage fehlerfrei ist und die Bearbeitung normal fortsetzen, bzw. bei fehlerhaften Anfragen in einen Subprozeß mit besonderer nachgelagerter Verarbeitung wechseln.*

Enthält ein State Node eine Transition mit einer *terminal*-Action, kann ein Request innerhalb einer Nutzer-Session diesen State Node als Quell-State-Node angeben, auch wenn dieser inkonsistent zum State Node in der Session ist. Nach Bearbeitung der Transition wird der State Node in der Session nicht auf den Ziel-State-Node der Transition geschaltet. Ein solcher Request wird – was die Workflowkonsistenz betrifft – sozusagen parallel zur aktuellen Session ausgeführt. Bezüglich der Anwendungsdaten arbeitet ein solcher Request jedoch auf den gleichen Sessiondaten.

In einer Web-Anwendung soll parallel zum aktuellen Workflow über einen Button ein neues Fenster geöffnet werden, in dem ebenfalls eine Anfrage an die zen Engine *gestellt wird. Der Einsprung der zweiten Anfrage erfolgt jedoch in einen anderen Teil des Workflows der Anwendung. Durch die Beantwortung dieser Anfrage soll sich der Workflow-Zustand des ursprünglichen Fensters nicht ändern. Zusätzlich operieren beide Anfragen auf dem gleichen Anwendungsdatenraum. Diese Situation kann mit einer* terminal-*Action gelöst werden.*

Operationen in Workflows

Es gibt mehrere äquivalente Möglichkeiten, Operationen mit Workflow-Elementen zu verbinden. Grundsätzlich kann workflowbasierte Geschäftslogik ohne Einschränkungen ausschließlich mit Transition Operations und Post Decision Operations modelliert werden. Hingegen dienen Pre State Operations, Post State Operations und Action Operations lediglich der Vereinfachung der Modellierung, weil sich so bestimmte Situationen im Modell zusammenfassen lassen.

Wenn jede eingehende Transition eines State Nodes mit der gleichen Operation belegt ist, können diese durch eine Pre State Operation zusammengefaßt werden. Wenn jede ausgehende Transition eines State Nodes mit der gleichen Operation belegt ist, genügt eine Post State Operation. Wird dagegen in allen Transitionen, die mit der gleichen Action bezeichnet sind, eine bestimmte Operation ausgeführt, bietet es sich an, diese durch eine Action Operation zu ersetzen.

7.1.3 Datenmodellierung

Atome

Atome, denen eine Domäne zugeordnet wurde, können zur Gestaltung von Selectboxen verwendet werden. Die Selectbox stellt den Domänen-Inhalt dar, das Atom enthält den ausgewählten Domänen-Wert.

Atome, die gleichzeitig als *mandatory* definiert sind, können Default-Values besitzen. Wird ein solches Atom ausgegeben, das noch nicht in der Session mit einem Dateninhalt existiert, wird stattdessen der Default-Value angezeigt. Besonders nützlich ist diese Eigenschaft zur Vorbelegung von Checkboxen oder Selectboxen.

Compositions

Compositions dienen zur Zusammenfassung von beliebigen anderen Elementen. Mit Compositions lassen sich Datenmodelle übersichtlich gliedern.

Listen

Listen aus Compositions können zur Gestaltung von HTML-Dateneingabe-Tabellen verwendet werden. Die Composition besteht aus Atomen und bestimmt so den Spaltenaufriß der Tabelle. Die Liste definiert die Tabellenzeilen.

Listen aus Atomen können zur Gestaltung von Multi-Selectboxen verwendet werden, sofern die Darstellung vom Stylesheet unterstützt wird (das Stylesheet *generic.xsl* stellt Listen aus Atomen nur als Multi-Selectbox dar, wenn das Atom zusätzlich mit dem Attribut *multi* gekennzeichnet wird). Die Liste sollte für Multi-Selectboxen unbedingt mit einer *default size* von 1 angelegt werden. Dem Atom der Liste wird im Modell eine Domäne zugeordnet. Diese definiert die auswählbare Wertemenge. Der Benutzer kann dann zur Laufzeit mehrere Einträge aus der Menge auswählen. Die Laufzeit-Liste enthält nach dem Absenden der Multi-Selectbox soviele Atome, wie Einträge aus der Menge ausgewählt wurden.

Ressourcen

Ressourcen dienen dazu, Daten- und Workflow-Elemente mit locale-spezifischen Texten, URL's, o.ä. zu versorgen, die in erster Linie zur Vertextung von HTML-Ausgaben verwendet werden. Dies sind typischerweise Seiten-Beschriftungen wie Überschriften, Texterläuterungen, Labels vor Eingabefeldern, Pfade für locale-spezifische Bilder wie z.B. Buttons oder Grafiken, die Text enthalten. Ressourcen können auch HTML-Tags enthalten, z.B. Formatierungsanweisungen, Links innerhalb eines Textes oder sogar JavaScript-Aufrufe. Hier sollte jedoch sorgfältig abgewägt werden, ob HTML-Tags nicht besser Bestandteil des XSL-Stylesheets sein sollten.

In einer HTML-Seite soll folgendes Formularfragment angezeigt werden

Tan [ ] Bitte nach Gebrauch von ihrer **TAN-Liste** streichen!

Dem Atom tan *wurden z.B. folgende zwei Ressourcen zugewiesen:*
label = Tan
note = Bitte nach Gebrauch von ihrer TAN-Liste streichen!

Normalerweise können Ressourcen immer an passenden State Nodes, Actions oder FOM-Elementen befestigt werden. Es ist im allgemeinen kein gutes Design, FOM-Elemente ohne Dateninhalt nur deswegen in das Datenmodell aufzunehmen, um sie mit Ressourcen auszustatten.

Ressourcen sollten normalerweise nicht über Operationen gefüllt oder gesetzt werden, sondern immer statisch im Modell definiert werden. Für dynamische Änderungen von Ressourcen-Inhalten, die von Geschäftslogik abhängig sind, können parametrisierte Ressourcen genutzt und mit dem Datenmodell verknüpft werden.

Attribute

Attribute dienen dazu, Daten- und Workflow-Elemente mit locale-unspezifischen Daten zu versorgen, die in erster Linie die Visualisierung von HTML-Ausgaben unterstützen. Dies können z.B. Darstellungshinweise wie Tabellenbreiten, Farben oder das Deaktivieren von Formular-Elementen sein. Insbesondere generische oder wiederverwendbare Stylesheets können so über Attribute spezifische Designdetails steuern. Inhalt und Funktion von Attributen können zwischen Anwendungs-Modellierer und Stylesheet-Autor frei bestimmt werden. Attribute können im Datenmodell statisch definiert werden und/oder von Operationen dynamisch gesetzt und verändert werden.

7.1.4 Operations

Argumente

Operationen arbeiten auf beliebig vielen Ein- und Ausgabe-Argumenten. Eingabe-Argumente, die auf Elementen basieren, werden in der Session identifiziert und an die Operation übergeben. Ausgabe-Argumente werden in die Session integriert, wobei evtl. noch nicht vorhandene Elternelemente mit angelegt werden.

Einfache Element-Argumentübergabe mit call by value*:*

```
public static Float calcSum(Float a, Float b) {
  return new Float(a.floatValue() + b.floatValue());
}
```

Einfache Element-Argumentübergabe mit call by reference *und einer* de.zeos.zen.api.fom.List:

```
public static Integer listSize(List l) {
  return new Integer(l.size());
}
```

Argumente können auch Superklassen verwenden. Der folgenden Operation können beliebige Subklassen von Number *übergeben werden:*

```
public static Boolean isPositive(Number n) {
  return new Boolean(n.doubleValue() > 0.0L);
}
```

Mehrere Ausgabeargumente werden mit Object-Arrays realisiert. Diese Operation gibt den Vor- und Nachnamen eines Kunden zurück. Die Kundeninformationen werden anhand der angegebenen Kundennummer in einer gesonderten Methode z.B. aus einer Datenbank geholt.

```
public static Object[] lookupCustomerName(Integer customerNo) {
  Customer cust = lookupCustomer(customerNo);
  return new String[] {cust.getFirstName(), cust.getLastName()};
}
```

Call With <null>

Mit der Eigenschaft *call with <null>* kann festgelegt werden, ob eine Operation mit Argumenten aufgerufen werden kann, die *null* sind. Bei *call by value* bezieht sich das auf Dateninhalte von Atomen, bei *call by reference* auf die übergebenen FOM-Elemente.

Standardmäßig sind Workflow Operations *call with <null>*, Business Rules dagegen nicht. In einigen Spezialfällen kann es sinnvoll sein, diese Option anzupassen. Bei Aktivierung muß man darauf achten, daß die Operation mit möglichen null-Argumenten umgehen kann.

Ist die Eigenschaft call with <null> *nicht aktiviert , kann dann sicher auf den übergebenen Parametern gearbeitet werden.*

```
public static Integer distanceBetween(Integer a, Integer b) {
  return new Integer(Math.abs(b.intValue() - a.intValue()));
}
```

Ist die diese Eigenschaft hingegen aktiviert, muß die Operation mit null-*Argumenten umgehen können. Hier soll z.B. ein Atom konstruiert werden, wenn es noch nicht in der Session vorhanden ist:*

```
public static Atom constructAtom(Atom a, String name) {
  if (a == null && name != null)
    a = ElementBuilder.getInstance().newAtom(name);
  return a;
}
```

OperationExceptions, ValidationRuleExceptions

Operations und Validation Rules signalisieren auftretende Benutzerfehler durch *OperationExceptions* bzw. *ValidationRuleExceptions*. Kann eine Operation mehrere unterschiedliche Benutzerfehler hervorrufen, muß beim Werfen der entsprechenden Exception ein Fehlerbezeichner mitgegeben werden, der zur Auswahl der passenden Fehlermeldung dient. Für jeden solchen Fehlerbezeichner einer Operation muß eine Fehlermeldung modelliert werden.

Hier eine Operation, die nur einen Benutzerfehler erzeugen kann. Daher wird kein Fehlerbezeichner angegeben, eine Fehlermeldung mit dem Standard-Bezeichner error *muß definiert sein:*

```
public static Integer distanceBetween(Integer a, Integer b)
                                        throws OperationException {
  int aVal = a.intValue();
  int bVal = b.intValue();
  if (bVal < aVal)
    throw new OperationException();
  return new Integer(bVal – aVal);
}
```

Diese Validation Rule kann zwei verschiedene ValidationRuleExceptions werfen, die mit unterschiedlichen Fehlerbezeichnern versehen werden:

```
public static void checkValidAge(Integer age) throws
ValidationRuleException{
  int ageVal = age.intValue();
  if (ageVal < 18)
    throw new ValidationRuleException("too-young");
  else if (ageVal > 64)
    throw new ValidationRuleException("too-old");
}
```

Business Rules

Eine Computation Rule oder Validation Rule wird ausgeführt, wenn mindestens ein Eingabe-Argument der Operation als Eingabe eines Requests vorkommt und, im Vergleich mit der Session, neu ist oder sich geändert hat. Computation Rules können über ihre Rückgabe-Argumente weitere Business Rules auslösen. Liegen nicht alle Eingabe-Argumente in der Session vor, wird die Operation nur ausgeführt, wenn sie als *call on <null>* definiert wurde.

In einem FOM-Baum ist eine Composition comp *mit den Atomen* a, b, c, d, e *als Kindelemente modelliert. Die Java-Methode*

```
public static Float increase(Float value, Float percentage) {
  return new Float(value.floatValue() * (1.0 + percentage/100.0));
}
```

wird *viermal als Computation Rule definiert und wie folgt mit den Atomen verbunden:*

```
b = increase1(a, 10.0)
d = increase2(b, 10.0)
c = increase3(a, d)
e = increase4(b, 10.0)
```

Liegt das Atom a *als neue bzw. geänderte Eingabe vor, ergibt sich folgende Ausführungsreihenfolge:*

```
1. increase1
2. increase2, increase4
3. increase3
```

Die Computation Rule increase3 *hängt von* increase1 *und* increase2 *ab, die Computation Rules* increase2 *und* increase4 *nur von* increase1. *Ob* increase2 *oder* increase4 *jeweils zuerst ausgeführt wird, hängt von ihrer definierten Priorität ab.*
Nach den Computation Rules werden alle Validation Rules ausgeführt, die mit den Atomen a, b, c, d, *oder* e *verknüpft sind. Die Ausführungsreihenfolge der Validation Rules hängt ausschließlich von ihrer Priorität ab.*

Bauen Computation Rules aufeinander auf, muß darauf geachtet werden, daß keine zirkulären Abhängigkeiten entstehen.

Erlaubt ist, wenn eine Computation Rule ein Eingabe- und Ausgabe-Argument unmittelbar auf das gleiche FOM-Element abbildet:

```
a = increase(a, 10.0)
```

Nicht erlaubt hingegen ist eine zirkuläre Abhängigkeit mit einer oder mehreren Zwischenstufen:

```
b = increase1(a, 10.0)
a = increase2(b, 10.0)
```

Listenoperationen

Operationen, die als Eingabe-Argument eine FOM-Liste haben, werden wie gewohnt einmal ausgeführt. Werden einer Operation jedoch Argumente zugeordnet, die unterhalb einer FOM-Liste liegen, wird die Operation so oft ausgeführt, wie die Liste Kindelemente hat.

Wir haben z.B. folgenden FOM-Baum in der Session, veranschaulicht durch die FOM-Pfadausdrücke:

```
/data/percentage = 5.0
/data/$items/item[0] = 10.0
/data/$items/item[1] = 20.0
/data/$items/item[2] = 15.0
```

Die Java-Methode increase *aus dem vorigen Beispiel wird nun als Operation folgendermaßen mit dem Datenmodell verknüpft:*

```
item = increase(item, percentage)
```

Die Operation wird dann dreimal folgendermaßen aufgerufen:

```
item[0] = increase(10.0, 5.0)
item[1] = increase(20.0, 5.0)
item[2] = increase(15.0, 5.0)
```

Argumente oberhalb der Liste, die Liste selbst eingeschlossen, bleiben konstant (im Beispiel also percentage*). Argumente unterhalb der Liste werden bei jedem Aufruf entsprechend weitergeschalten (im Beispiel* item*).*

Die Ausführung von Operationen, deren Eingabe- und/oder Rückgabe-Argumente innerhalb einer Hierarchie unterhalb von beliebig vielen FOM-Listen liegen, funktioniert analog. Nicht erlaubt ist, wenn eine Operation Eingabe-Argumente unterhalb von benachbarten Listen hat.

Bei folgendem FOM-Baum

```
/data/percentage = 5.0
/data/$items/item[0]/$lower-items/lower-item[0] = 10.0
/data/$items/item[0]/$lower-items/lower-item[0] = 10.0
/data/$items/item[0]/percentage = 3.0
/data/$items/item[1]/$lower-items/lower-item[0] = 20.0
/data/$items/item[1]/$lower-items/lower-item[1] = 30.0
/data/$items/item[1]/$lower-items/lower-item[2] = 15.0
/data/$items/item[1]/percentage = 7.0
/data/$parallel-items/parallel-item[0]/p = 33.0
/data/$parallel-items/parallel-item[1]/p = 66.0
```

ist die Definition von nachfolgenden Operationen legal:

```
lower-item = increase(lower-item, /data/percentage)
lower-item = increase(lower-item, /data/$items/item/percentage)
```

Folgende Operation ist jedoch nicht legal, da sie benachbarte Listen-Elemente nutzt:

```
lower-item = increase(lower-item, /data/$parallel-items/parallel-item/p)
```

Aggregationen

Wird eine Listenoperation als *aggregation* definiert, wird ihre Java-Klasse vor der mehrfachen Ausführung nur einmal instantiiert. Die Methode wird anschließend wie eine normale Listenoperation so oft aufgerufen, wie die Liste Kindelemente hat. Sie kann jedoch über den mehrfachen Aufruf hinweg in der Instanz einen Zustand mitführen. Am Ende wird nur ein Ergebnis von der Instanz zurückgegeben. Dieses wird normalerweise in ein Element oberhalb der Liste geschrieben.

Der nachfolgende FOM-Baum liegt in der Session:

```
/data/sum = <null>
/data/$items/item[0] = 10
/data/$items/item[1] = 20
/data/$items/item[2] = 15
```

Die Java-Klasse für eine einfache Aufsummierung sieht z.B. so aus:

```
public class Sum implements AggregationOperation {
    private int sum = 0;
    public void add(Integer i) {
        this.sum += i.intValue();
    }
    public Object getResult() {
        return new Integer(this.sum);
    }
}
```

Sie wird folgendermaßen mit dem Datenmodell verknüpft:

```
sum = add(item)
```

Durch den mehrfachen Aufruf der Operation add *werden alle Listen-Elemente aufsummiert und das Ergebnis in das Atom* sum *geschrieben.*

7.1.5 Domänen

Bei der Definition einer entwicklerdefinierten Domäne wird auch der *lifecycle* der Domäne festgelegt. Dieser bestimmt, über welchen Zeitraum eine Domäne eine konstante Key/Value-Menge besitzt.

Eine *system*-Domäne hat über die gesamte Lebensdauer des Repository-Caches und für alle Benutzer eine feste Key/Value-Menge.

Beispiele für eine system-*Domäne: Eine Auswahlliste von Ländern, die aus einer Datenbank gelesen wird oder eine Auswahlliste von Zahlen, die über ein Konstanten-Argument definiert wird. Für letztere könnte eine einfache Implementierung so aussehen:*

```
public class Numbers implements Domain {
    private int max;
    public Numbers(Integer max) {
        this.max = max.intValue();
    }
    public boolean containsKey(String key, Locale locale) {
        return (Integer.parseInt(key) <= max);
    }
    public Iterator getKeys(Locale locale) {
        return new Iterator() {
            int i = 0;
            public Object next() {
                if (i > max)
                    throw new NoSuchElementException();
                return Integer.toString(i++);
            }
            public boolean hasNext() {
                return (i <= max);
            }
            public void remove() {
                throw new UnsupportedOperationException();
            }
        };
    }
    public String getValue(String key, Locale locale) {
        String value = key;
        // nur für max. zweistellige Zahlen!
        if (value.length() == 1)
            value = "0" + value;
        return value;
    }
}
```

Eine *session*-Domäne hat dagegen nur über die Lebensdauer einer Benutzer-Session eine feste Key/Value-Menge. Mit einer *session*-Domäne können verschiedene Benutzer mit eigenen, aber jeweils festen Domänen-Inhalten versorgt werden, z.B. abhängig von einem Benutzerprofil.

Domänen, deren Inhalte sogar während einer Benutzer-Session variabel sein sollen, werden als *state*- oder *transition*-Domänen definiert. Diese werden dann pro Request neu instantiiert, wobei *state*-Domänen von Ausgabe bis zur nächsten Eingabe konstant bleiben, *transition*-Domänen von der Eingabe bis zur Ausgabe.

Beispiel für eine state-*Domäne: In einem Währungsrechner sollen Euro-Beträge in eine andere Währung umgerechnet werden. Die Zielwährung wird in einer Selectbox ausgewählt, diese wird von einer Domäne mit Inhalten gefüllt. Die Menge der Zielwährungen kann sich verändern, je nachdem, ob zur Zielwährung Kurse vorliegen oder nicht. Die Zielwährungen (und Kurse) werden unmittelbar vor ihrer Ausgabe/Anzeige aus der Datenbank geholt. Der Benutzer wählt eine Zielwährung aus und bekommt den umgerechneten Betrag zum Kurs der Zielwährung angezeigt.*
Beispiel für eine transition-*Domäne: In einer Anwendung zum Verkauf von Waren werden die aktuell vorhandenen Waren einer bestimmten Gattung in einer Selectbox dargestellt. Der Benutzer sucht sich die gewünschte Ware aus. Beim Absenden des Requests wird die Domäne neu aufgebaut. Hier kann gleich bestimmt werden, ob die Ware immer noch verfügbar ist oder zwischenzeitlich bereits verkauft wurde. Ist letzteres der Fall, wird automatisch die modellierte Fehlermeldung ausgegeben und in der Selectbox erscheinen die noch übrigen Waren.*

Durch die Verknüpfung von Domänen-Konstruktoren mit Argumenten lassen sich bei *session-, state-* und *transition-*Domänen auch Sessiondaten zur Konstruktion von Inhalten nutzen. Hier ist jedoch darauf zu achten, daß die benötigten Daten bei erstmaliger Verwendung der Domäne bereits in der Session vorliegen. Sessiondaten können z.B. als Entscheidungs- oder Filterkriterium für den Domänenaufbau genutzt werden, eine Domäne kann aber auch direkt aus Sessiondaten bestehen, z.B. aus den Inhalten einer FOM-Liste.

7.1.6 Datentypen

Die Dateninhalte von Atomen liegen als Java-Objekte bestimmter Klassen vor, die durch den Datentyp des Atoms definiert werden. Dies können generische Klassen sein wie z.B. *java.lang.String*, *java.lang.Integer* oder *java.util.Date*, aber auch geschäftsspezifische Klassen, wie z.B. die Repräsentation einer internationalen Kontonummer (IBAN). Konvertierungsklassen dienen zur eindeutigen Konvertierung dieser Objekte von und in Stringrepräsentationen. Datentypen können über Werfen einer *ParsingException* eine Validierung von Stringrepräsentationen vornehmen. Für differenzierte Validierungen sollten Business Rules verwendet werden.

Beispiel für einen selbstdefinierten Datentyp. Hier wird eine Versicherungsnummer mittels Datentyp auf Korrektheit überprüft und in java.lang.Long *konvertiert. Die Locale findet in diesem Beispiel keine Beachtung, die triviale Formatierung ruft* toString() *auf:*

```
public class InsuranceNoConversion implements DataConversion {
    public Object parse(String string, Locale locale)
                                           throws ParsingException {
        Long l;
        long lval;
        try {
            l = new Long(string);
            lval = l.longValue();
        }
        catch (NumberFormatException e) {
            throw new ParsingException();
        }
        if (lval < 100000000l || lval > 499999999l)
            throw new ParsingException();
        return l;
    }
    public String format(Object o, Locale l) {
        return o.toString();
    }
}
```

7.1.7 Stylesheets

Das generische Stylesheet *generic.xsl* bietet nur eine sehr eingeschränkte Darstellung für HTML-Seiten, die für den Produktivbetrieb in der Regel nicht ausreichend ist. Für eine individuelle Visualisierung muß daher ein eigenes Stylesheet erstellt werden. Dies gilt insbesondere auch für andere Ausgabeformate, z.B. PDF. Für Aufbau und konzeptionellen Ansatz gibt es durch die *zen Engine* keine Einschränkungen. In eigenen Stylesheets kann das Komponenten-Stylesheet *components.xsl* verwendet werden, es kann ein eigenes Komponenten-Stylesheet entwickelt werden, es können auch alle HTML-Komponenten direkt in der konkreten Visualisierung definiert werden. Stylesheets können monolithisch oder über Import- und Include-Anweisungen beliebig modular aufgebaut werden.

Generische Stylesheets

Stylesheets können so generisch konstruiert werden, daß sie beliebige XML-Rückgaben verarbeiten können. Dieser Ansatz ist im Stylesheet *generic.xsl* skizziert, kann aber natürlich in einem eigenen Stylesheet adaptiert und perfektioniert werden. Neben Attributen, die über die Anwendungseigenschaften *Include FOM Types, Include Datatypes* und *Include Read Only* gesteuert werden, lassen sich beliebige andere Attribute definieren, die das Design innerhalb des Stylesheets steuern. Als Beispiel dient das Attribut *multi*, das im Stylesheet *generic.xsl* bei Ausgabe einer Liste von Atomen zwischen einer Visualisierung als Multi-Selectbox (wenn das Attribut vorhanden ist) und einer Visualisierung einer einspaltigen Tabelle (wenn das Attribut nicht vorhanden ist) unterscheidet. Natürlich könnte auch im Stylesheet mit Hilfe von XSL-Ausdrücken eine Liste von Atomen erkannt werden und dieser Modellfall grundsätzlich als Multi-Selectbox visualisiert werden, wenn einspaltige Tabellen ohnehin keine Rolle spielen.

Werden Attribute zur Steuerung der Visualisierung verwendet, müssen diese zwischen dem Ersteller des Daten- bzw. Workflow-Modells und dem Ersteller des Stylesheets genau festgelegt werden. Mit entsprechend vielen Attributen, die jeweils spezifische Anzeigedetails steuern, kann eine einheitliche Oberfläche für eine ganze Gruppe von Anwendungen mit ähnlichem Design realisiert werden. Im Extremfall kann über eine Vielzahl von definierten Attributen ein pixelgenaues Design über ein generisches Stylesheet realisiert werden, das sogar für verschiedene Anwendungen mit unterschiedlichen Designs verwendet werden könnte. Je mehr Attribute jedoch definiert werden, desto komplexer und schwieriger ist die Pflege des Modells. Wo die Grenze liegt, ist je nach Analyse bzw. Gewichtung von Vor- und Nachteilen im Einzelfall zu entscheiden.

Generische Stylesheets können mit Modell-Zusatzinformationen ein Design für eine ganze Reihe von Anwendungen mit mäßig komplexem Layout realisieren. Sie sind absolut wiederverwendbar, verlagern jedoch Komplexität zur Layoutgestaltung durch die Menge der nötigen Attribute in die Anwendungsmodelle.

Anwendungsspezifische Stylesheets

Einen grundsätzlich anderen Ansatz bietet die Erstellung von Stylesheets, die jeweils nur eine individuelle Anwendung visualisieren. Sie sind dann vom Aufbau her auf den konkreten FOM-Baum der Anwendung ausgerichtet. Die Vergabe von Attributen zur Steuerung des Designs ist praktisch nicht nötig, die Templates des Stylesheets berücksichtigen in ihren *match*-Patterns stattdessen die konkreten FOM-Elemente. Auch hier sind mehrere Ansätze denkbar. So können seitenbasierte Stylesheets entwickelt werden, die z.B. für jeden State Node ein individuelles Design festlegen. Workflowbasierte Stylesheets können anhand des Vorkommens von konkreten FOM-Elementen – unabhängig vom State Node – Anzeigeblöcke definieren, die zusammengenommen die Visualisierung bilden. Öfter vorkommende Anzeigeblöcke können z.B. in eigene Stylesheets ausgelagert werden.

Anwendungsspezifische Stylesheets können ohne Modell-Zusatzinformationen ein optimales und beliebig komplexes Design für eine Anwendung realisieren. Allerdings sind sie kaum wiederverwendbar. Durch modulare Stylesheet-Gestaltung können ggf. wiederverwendbare Teilbereiche extrahiert werden.

XSL-FO

Für die Generierung von PDF's wird XSL-FO eingesetzt. Die Vorgehensweise ist grundsätzlich ähnlich wie im HTML- oder XML-Umfeld. Auch hier ist ein generischer Ansatz denkbar. Durch die detaillierten Designanweisungen ist die Problematik bezüglich einer Modellüberfrachtung durch Attribute aber noch wesentlich kritischer zu sehen. Dennoch sollte diese Alternative bei der Entscheidung mitberücksichtigt werden.

7.2 Patterns

Nachfolgend werden komplexere Muster beschrieben, wie sie in realen Anwendungen häufig auftreten können. Diese Muster umfassen die Modellierung, die Java-Implementierung und zum Teil auch die Stylesheet-Implementierung.

7.2.1 Ein- und Ausgabe mit *In-Opt* und *Out-Opt*

Aufgabenstellung

Eingabedaten sollen abhängig von Sessiondaten fallweise Bestandteil oder kein Bestandteil der Ein- bzw. Ausgabe sein.

Beispiel

Eine formularbasierte Anwendung für einen Finanzierungsrechner besteht aus zwei Seiten. Auf der ersten Seite kann der Kunde notwendige Angaben zum gewünschten Darlehen machen, auf der zweiten Seite wird das Finanzierungsangebot angezeigt.

Die nötigen Angaben auf der ersten Seite sind zunächst die gewünschte Darlehenssumme und Laufzeit. Mit einem Button kann das Formular um weiteren Angaben zu einer Kreditversicherung erweitert werden. In diesem Fall wird in dem gleichen Formular noch zusätzlich das Geburtsdatum des Kunden abgefragt. Bei nochmaligem Klick auf den Button werden die Angaben zur Kreditversicherung wieder ausgeblendet.

Lösung

Da das Geburtsdatum im Falle der gewünschten Kreditversicherung Teil der Ein- und Ausgabe sein kann, muß das Geburtsdatum für den State Node des Formulars als *in* und *out* definiert werden. Ohne Kreditversicherung ist das Geburtsdatum nicht Teil der Ein- und Ausgabe, daher muß es zusätzlich als *in-opt* und *out-opt* definiert werden.

Das Eingabefeld für das Geburtsdatum wird in dieser Einstellung angezeigt, wenn das entsprechende FOM-Atom Bestandteil der Sessiondaten ist, ansonsten wird es nicht angezeigt. Das FOM-Atom muß also fallweise in der Session erzeugt bzw. gelöscht werden. Hierzu definieren wir folgende Java-Methode als Operation:

```
public static Atom toggleAtom(Atom a, String name) {
    if (a == null)
      a = ElementBuilder.getInstance().newAtom(name);
    else
      a = null;
    return a;
}
```

Wir ordnen der Operation das FOM-Atom für das Geburtsdatum mit der Eigenschaft *call by reference* als Ein- und Ausgabe-Argument zu. Zusätzlich definieren wir eine Konstante mit dem Namen dieses FOM-Atoms. Wird in der Methode das FOM-Atom für das Geburtsdatum neu angelegt, wird der Inhalt dieser Konstante als Name für das Atom verwendet. Diese

Konstante ist also das zweite Eingabe-Argument der Operation. Damit die Operation später auch aufgerufen wird, wenn das FOM-Atom noch nicht in der Session existiert, wird die Operation zusätzlich mit der Eigenschaft *call with <null>* versehen.

Natürlich könnte der Name für das Geburtsdatums-Atom direkt im Java-Code angegeben werden. So können wir aber den Namen des Atoms ohne Anpassungen des Java-Codes jederzeit im Repository ändern und wir haben eine Java-Methode geschrieben, die in ähnlichen Aufgabenstellungen immer wieder eingesetzt werden kann.

Für den Button, mit dem die Kreditversicherung ein- und ausgeschaltet werden soll, wird eine Action definiert. Wird die Action als *default* definiert, wird das Geburtsdatum für die Kreditversicherung nur eingeblendet, wenn das übrige Formular korrekt ausgefüllt ist. Soll das Geburtsdatum jedoch unabhängig vom Zustand des Formulars immer ein- und ausgeblendet werden können, definieren wir die Action als *nonvalidating*.

Diese Action wird nun einer Transition zugeordnet, die den State Node des Formulars reflexiv mit sich selbst verbindet. Nun kann die Operation entweder an die Action oder an die Transition gehängt werden. Das Modell ist damit fertig.

Weitere Anmerkungen

Auf die gleiche Weise können natürlich auch FOM-Elemente angelegt bzw. gelöscht werden, die nicht auf der gleichen Formularseite als Ein- und Ausgabe definiert sind.

In diesem Beispiel kann bei der Berechnung des Kredites die Frage, ob der Kunde eine Kreditversicherung wünscht oder nicht, mit der Existenz des FOM-Atoms für das Geburtsdatum entschieden werden. Dies ist natürlich ein sehr impliziter Zusammenhang. Expliziter wäre es, ein eigenes FOM-Atom für Auswahl der Kreditversicherung zu definieren, das nicht in der Ein- bzw. Ausgabe auftaucht, sondern gleichzeitig mit dem Ein- und Ausblenden des Geburtsdatums geschaltet wird. Eine entsprechende Operation kann vor oder nach der ersten Operation in der gleichen Action bzw. Transition eingetragen werden.

Damit die Action je nach Ein- bzw. Ausschalten der Kreditversicherung einen anderen Text darstellt bzw. eine andere Grafik benutzt, kann die Action-Ressource z.B. mit dem Geburtsdatum oder dem soeben angesprochenen expliziten Kreditversicherungs-Atom parametrisiert werden.

Bei einem Submit-Button unter Verwendung des Geburtsdatums könnte die Ressource zur Beschriftung wie folgt aussehen:

```
„Kreditversicherung {0,choice,null#ein|null<aus}“
```

Bei einem Image-Button unter Verwendung eines expliziten FOM-Atoms für die Kreditversicherung auf Boolean-Basis könnte die Ressource wie folgt aussehen:

```
„/images/kredit_{0,choice,false#ja|true#nein}.gif“
```

7.2.2 Data Grids in HTML

Aufgabenstellung

Realisierung einer Tabelle aus Eingabe-Elementen wie z.B. Texteingabefeldern, Checkboxen, Selectboxen, u.ä. (Data Grid).

Beispiel

In einer Tabelle sollen Personendaten eingegeben werden. In jeder Zeile werden Details zu je einer Person erfaßt. Die Tabellenspalten bestehen aus einer Selectbox, in der die Anrede einer Person auswählbar ist, und Eingabefeldern für Vorname, Nachname und Geburtsdatum.

Lösung

Im Datenmodell wird eine FOM-Liste für die Personen definiert. Unter diese Liste wird eine Composition für eine Person eingehängt, die aus je einem Atom für Anrede, Vorname, Nachname und Geburtsdatum besteht. Dem Atom für die Anrede wird eine Domäne mit den möglichen Anreden zugewiesen. Diese Liste wird nun auf einem State Node als Ein- und Ausgabe markiert. Damit bei der erstmaligen Anzeige dieses State Nodes auch bei leerer Session leere Listen-Elemente angezeigt werden können, wird der Liste eine *default size* zugewiesen. Das Data Grid wird nun mindestens mit der entsprechenden Zeilenanzahl dargestellt.

Anmerkung

Das mitgelieferte generische Stylesheet *generic.xsl* stellt Data Grids standardmäßig dar, die Vorgehensweise zur Konstruktion von Data Grids ist jedoch unabhängig vom Stylesheet.

Erweiterung: Key Elemente

Wird nun beispielsweise der Vor- und Nachname als Pflichtfeld (*mandatory*) markiert, kann die Seite mit dem Data Grid zunächst nur abgeschickt werden, wenn diese Felder in allen Zeilen ausgefüllt werden. In der Praxis ist es aber oft wünschenswert, auch weniger Personen einzutragen. Andererseits sollen die einzelnen Tabellenzeilen auch nicht partiell ausgefüllt werden dürfen.

Um diese Problematik zu lösen, können Atome als *key element* markiert werden: Wird in einer Tabellenzeile kein einziges Atom ausgefüllt, das als *key element* markiert ist, gilt die komplette Zeile als nicht ausgefüllt und wird verworfen. Dabei werden Fehlerzustände, die in einer solchen Zeile vorkommen, z.B. nicht ausgefüllte *mandatory*-Felder, ebenfalls verworfen.

Verworfene Zeilen werden zusätzlich aus der Session gelöscht. Damit lassen sich auch Zeilen effektiv löschen, wenn ihre als *key element* markierten Eingabefelder vom Benutzer gelöscht werden.

Untersuchen wir folgende Beispiel-Kombinationen aus mandatory *und* key element:

1. *Vor- und Nachname* mandatory, *kein* key element
   - ⇒ *Vor- und Nachname aller Tabellenzeilen müssen gefüllt werden*
2. *Vor- und Nachname* mandatory, *beide* key element
   - ⇒ *Sind weder Vor- noch Nachname gefüllt, wird die Zeile als nicht ausgefüllt gewertet, selbst wenn z.B. ein Geburtsdatum eingetragen wurde. Die Zeile wird komplett verworfen. Wird nur der Vorname ausgefüllt, gilt der leere Nachname als Verletzung der* mandatory-*Eigenschaft. Umgekehrt analog.*
3. *Vor-, Nachname und Geburtstag* mandatory, *nur Vor- und Nachname* key element
   - ⇒ *wie 2., nur muß nun zusätzlich noch das Geburtsdatum angegeben werden, wenn Vor- und Nachname ausgefüllt sind*
4. *Alle Atome* mandatory *und* key element
   - ⇒ *Tabellenzeilen müssen immer ganz oder gar nicht ausgefüllt werden*

Erweiterung: Dynamische Veränderung des Data Grids

Wird die Liste des Data Grids mit einer *default size* 4 definiert, werden immer mindestens 4 Einträge angezeigt, maximal jedoch so viele, wie in der Session vorhanden sind. Sind weniger als 4 Listen-Einträge in der Session vorhanden, werden die restlichen Einträge im Data Grid leer angezeigt.

Wir wollen dem Benutzer nun erlauben, das Data Grid dynamisch vergrößern. Hierzu wird auf der Formularseite mit dem Data Grid ein Button angebracht, mit dem das Formular um 4 zusätzliche Einträge erweitert werden kann. Wir definieren eine Java-Methode als Operation:

```
public static void increaseList(List list, String name, Integer size) {
    ElementBuilder builder = ElementBuilder.getInstance();
    for (int i = 0; i < size.intValue(); i++) {
      Composition listElement = builder.newComposition(name);
      list.addElement(listElement);
    }
  }
```

Als Argumente verbinden wir mit der Operation die Personen-Liste, den Namen der Personen-Composition und die gewünschte Listen-Vergrößerung, in diesem Fall 4. Diese Operation wird nun mit dem Button verknüpft. Dazu wird die Action auf eine Transition gelegt, die den State Node des Formulars mit sich selbst verbindet.

Wird nun auf den Button geklickt, werden der Liste 4 Elemente hinzugefügt. Im Zusammenhang mit der Markierung von Atomen als *key element* ist zu berücksichtigen, daß nicht ausgefüllte Zeilen automatisch gelöscht werden. Wird in dem Data Grid also eine Person eingetragen und auf den Button geklickt, werden die übrigen 3 Zeilen gelöscht, jedoch 4 neue Einträge hinzugefügt, so daß insgesamt 5 Einträge angezeigt werden. Sollen dagegen bei Klick auf den Button immer 8 Einträge erscheinen, muß die Operation noch die *default size* berücksichtigen. Diese könnte als weiteres Argument übergeben werden.

Schließlich noch eine verwandte Aufgabenstellung: Das Data Grid soll nicht direkt vergrößert werden, stattdessen wird die Anzahl der Zeilen des Data Grids auf einer vorgelagerten Formularseite angegeben. Das Data Grid wird an dieser Stelle durch eine Operation auf die entsprechende Anzahl vergrößert (oder auch verkleinert). Hier muß berücksichtigt werden, daß eventuell die gesamte Liste noch nicht in der Session vorliegt. Wenn in der Operation die übergebene Liste also *null* ist, muß diese erst angelegt werden, bevor sie mit Listen-Einträgen gefüllt werden kann. Damit die Liste in den FOM-Baum integriert werden kann, muß die Operation die Liste als Rückgabewert modellieren.

7.2.3 Business Rules auf programmatisch gefüllten Eingabefeldern

Aufgabenstellung

Ein Formular wird von einer Operation mit speziellen editierbaren Daten vorbelegt. Auf den Daten sind Business Rules definiert.

Beispiel

Der Benutzer kann in einem Formular Überweisungsvorlagen anlegen. Die entsprechenden Daten werden in einer Datenbank gespeichert. Soll die Vorlage angepaßt werden, werden die Daten aus der Datenbank in das Formular geladen. Auf der Ziel-Bankleitzahl der Überweisungsvorlage ist eine Computation Rule definiert, die automatisch den Namen des Kreditinstitutes auflöst.

Lösung

Business Rules werden genau dann ausgelöst, wenn ihre Eingabe-Argumente auf Daten basieren, die entweder neu sind oder sich im Vergleich zu den Sessiondaten geändert haben. Die Daten für die Überweisungsvorlage werden in einer Operation aus der Datenbank geholt und direkt in der Session gespeichert. Wird die Ziel-Bankleitzahl im Eingabeformular nicht verändert, ist sie im Vergleich zur Session weder neu noch geändert. Daher wird die Computation Rule nicht ausgelöst.

Die Ziel-Bankleitzahl kann durch die Operation in der Session daher zusätzlich mit dem Attribut *builtin:dirty* versehen werden. Damit gilt sie beim Abschicken des Formulars als geändert und kann Business Rules auslösen.

7.2.4 Listen mit Blätterlogik

Aufgabenstellung

Eine Tabelle bzw. ein Data Grid soll nur ausschnittsweise auf einer Seite angezeigt werden. Der Ausschnitt soll vor- und zurückgeblättert werden können.

Beispiel

Der Benutzer kann sich alle getätigten Kontobuchungen ansehen. Da die Liste der Kontobuchungen sehr lang werden kann, wird nur immer ein Ausschnitt von 10 Buchungen auf einer Seite angezeigt. Dieser Ausschnitt wird über Buttons vor- und zurückgeblättert.

Lösung

Ein FOM-Listen-Element kann in Operations mit den Attributen *builtin:from* und *builtin:to* versehen werden. Bei der XML-Ausgabe wird nur der entsprechende Bereich der Liste ausgegeben. Falls es sich um ein Data Grid handelt, die Liste also aus Eingabefeldern besteht, muß das XSL-Stylesheet sicherstellen, daß die von der HTML-Seite gesendeten Listen-Daten ebenfalls mit den Attribute *builtin:from* und *builtin:to* versehen werden, um den gesendeten Listen-Ausschnitt richtig in die Session einpassen zu können. Dies stellt z.B. das *path*-Template des Komponenten-Stylesheets *components.xsl* sicher.

In unserem Beispiel werden die Kontobuchungen über eine Operation aus der Datenbank geholt. Hierzu wird die Liste *buchungen* erzeugt, für jede Buchung wird eine Composition *buchung* mit den Atomen *datum*, *text* und *umsatz* angelegt und mit den Werten aus der Datenbank gefüllt. Der anzuzeigende Listen-Ausschnitt muß nun festgelegt werden. Um diesen flexibel handhaben zu können, definieren wir hierfür eine eigene Operation, die wir hinter der obigen Operation z.B. in die gleiche Transition hängen. Die Java-Methode dafür setzt die Attribute wie folgt, der Wert *pageSize* bestimmt dabei die Größe des anzuzeigenden Listen-Ausschnittes:

```
public static void setListPageSize(List list, Integer pageSize) {
    list.setAttribute("builtin:from", "0");
    list.setAttribute("builtin:to", Integer.toString(pageSize.intValue() –
1));
  }
```

Für die Buttons zum Blättern benötigen wir noch zwei Operationen, um über die Attribute den Listen-Ausschnitt vor- und zurückzuschalten. Überschreitet wie in der unten angegebenen Operation *forward* das Attribut *builtin:to* die tatsächliche Listengröße, werden leere Einträge erzeugt. Dies kann für Data Grids nützlich sein, da sie so weiter vergrößert werden können. Ist dies nicht erwünscht, muß das Attribut *builtin:to* einfach auf die Listengröße – 1 beschränkt werden.

```
public static void forward(List list, Integer stepSize) {
    int size = l.size();
    int step = stepSize.intValue();
    int from = Integer.parseInt(l.getAttribute("builtin:from"));
    int to = Integer.parseInt(l.getAttribute("builtin:to"));
    if (from + step < size) {
      l.setAttribute("builtin:from", Integer.toString(from + step));
      l.setAttribute("builtin:to", Integer.toString(from + step + step –
1));     }
  }

  public static void backward(List l, Integer stepSize) {
    int step = stepSize.intValue();
    int from = Integer.parseInt(l.getAttribute("builtin:from"));
    int to = Integer.parseInt(l.getAttribute("builtin:to"));
    if (from – step >= 0) {
      l.setAttribute("builtin:from", Integer.toString(from – step));
      l.setAttribute("builtin:to", Integer.toString(from – 1));
    }
  }
```

Anmerkung

Das mitgelieferte generische Stylesheet *generic.xsl* stellt Ausschnitte von Data Grids standardmäßig dar, die Vorgehensweise zur Konstruktion von Data Grids ist jedoch unabhängig vom Stylesheet.

Weitere Anmerkungen

Werden die ausgegebenen Listen-Elemente zusätzlich als Eingabedaten definiert, so muß mit dem Listen-Ausschnitt zusätzlich das Listen-Attribut *builtin:from* vom Stylesheet an die *zen Engine* zurückgesendet werden. Dieses darf jedoch niemals größer als die Listengröße in der Session sein. Anderenfalls wird das Attribut auf die Listengröße der Session zurückgesetzt. Dies hat bei der Verschmelzung von Request- und Sessiondaten zur Folge, daß Listen-Ausschnitte, die an die *zen Engine* gesendet werden, innerhalb der Sessionliste eingepaßt werden oder direkt an das Ende der Sessionliste angrenzen. Es kann also nie eine Lücke zwischen gesendetem Listen-Ausschnitt und Sessionliste entstehen.

7.2.5 Ausblendbare Buttons

Aufgabenstellung

Eine Action soll abhängig von der Geschäftslogik aktiviert bzw. deaktiviert werden können.

Beispiel

Das Beispiel aus 7.2.4 Listen mit Blätterlogikwird folgendermaßen erweitert: Am Anfang der Liste wird der Button zum Zurückblättern nicht angezeigt, am Ende der Liste wird der Button zum Vorwärtsblättern ausgeblendet.

Lösung

Das Deaktivieren von Buttons kann derzeit nur implizit gelöst werden, da eine Operation nicht auf Actions zugreifen kann. Wir können hierzu aber eine parametrisierte Action-Ressource nutzen. Zusätzlich wird ein FOM-Atom als Entscheidungskriterium definiert: Es kann z.B. die Werte *anfang*, *mitte* und *ende* einnehmen. Diese Werte werden in einer Operation, die z.B. als Pre State Operation auf dem Blätter-State-Node definiert werden kann, durch den Rückgabewert ins Atom gesetzt.

```
public static String pageStatus(List l) {
  int size = l.size();
  int from = Integer.parseInt(l.getAttribute("builtin:from"));
  int to = Integer.parseInt(l.getAttribute("builtin:to"));
  if (from == 0)
    return "anfang";
  else if (to >= size)
    return "ende";
  else
    return "mitte";
}
```

Das Atom wird dann als Argument der Ressource für den Vorwärts- bzw. Zurückbutton zugeordnet:

Vorwärtsbutton: „`{0,choice,"anfang"#/img/vorwaerts.png|"ende"#}`“
Rückwärtsbutton: „`{0,choice,"ende"#/img/rueckwaerts.png|"anfang"#}`“
Da der String mitte *nicht in der Ressource vorkommt, wird er auf* $-\infty$ *abgebildet. Somit wird die erste Alternative gewählt.*

Im Stylesheet kann der Button bei leerer Ressource ausgeblendet werden:

```
<xsl:choose>
  <xsl:when test="@resource:src != ''">
    <xsl:call-template name="input-image"/>
  </xsl:when>
  <xsl:otherwise>
    <!-- ausgeblendet -->
  </xsl:otherwise>
<xsl:choose>
```

Das generische Stylesheet *generic.xsl* blendet bei zugeordneter Ressource mit leerem Inhalt den Button für die Action aus.

7.2.6 Asynchroner Prozeß / Managed Message

Aufgabenstellung

Eine langdauernde Operation wird als asynchroner Prozeß gestartet. Der Benutzer kann auf einer Feedback-Seite sehen, ob der Prozeß bereits beendet ist. Die Feedback-Seite wird hierzu in regelmäßigen Abständen neu aufgebaut.

Beispiel

Zur Erzeugung eines Kontoauszugs wird in einer Bankanwendung ein externes Reporting-Tool angebunden. Dieses generiert den Kontoauszug und speichert den Kontoauszug in einer Datenbank ab. Danach kann der Kontoauszug aus der Datenbank gelesen und angezeigt werden. Da dieser Prozeß zeitaufwendig ist, soll dem Benutzer zwischenzeitlich der Status des Generierungsprozesses auf einer Feedback-Seite angezeigt werden. Die Feedback-Seite wird in regelmäßigen Abständen neu aufgebaut, bis der Prozeß beendet ist. Daraufhin wird der fertige Kontoauszug angezeigt.

Lösung

Der Kontoauszug wird über einen Button angefordert. Die zugehörige Transition verzweigt auf die Feedback-Seite und startet den asynchronen Prozeß, der den Zugriff auf das Reporting Tool implementiert. Der asynchrone Prozeß selbst wird als Managed Message umgesetzt. Die Implementierung erbt dazu von der Klasse *de.zeos.scf.component.ManagedMessageDefImpl*. Ein Message Bean *ReportGenerator* könnte z.B. so aussehen:

```
public class ReportGenerator extends ManagedMessageDefImpl {

    protected String getName() {
        return „ReportGenerator“;
    }
    protected void processMessage(MessagingData data) {
        // Erstellen des Kontoauszugs und speichern in der Datenbank
        // ...
        // Es gab keine Fehler
        this.setStatus(MessageStatus.FINISHED, data);
    }
}
```

Das Message Bean wird anschließend mit seinem Namen in der SCF-Konfiguration eingetragen. Die Java-Operation zum Aufruf dieses Prozesses könnte dann z.B. so aussehen:

```
public static void printReport(Integer account, Date from, Date to)
       throws OperationException {

       HashMap map = new HashMap();
       map.put(ACCOUNT, account);
       map.put(FROM, from);
       map.put(TO, to);
       ManagedMessanger mm = null;
       try {
           ServiceConnector sc = new InitialServiceConnector();
           mm = sc.getMessagingService().getManagedMessenger
(„ReportGenerator");
           mm.setMessageData(map);
           mm.startMessaging();
       }
       catch (Exception e) {
           throw new OperationException();
       }
   }
```

Die Feedback-Seite enthält einen automatischen Refresh. Dieser kann z.B. in JavaScript realisiert werden, aber auch über ein HTML-Meta-Tag. Der entsprechende Aufruf wird im Stylesheet kodiert. Das HTML-Meta-Tag ist in XSL etwas umständlich zu formulieren. Anwendungsspezifische Nutzdaten können analog an die URL gehängt werden:

```
(...)

     <xsl:text disable-output-escaping="yes">
       &lt;meta http-equiv="Refresh" content="3; URL=
     </xsl:text>
     <xsl:call-template name="refresh-url"/>
     <xsl:text disable-output-escaping="yes">"/></xsl:text>

   (...)

   <xsl:template name="refresh-url">
     <xsl:value-of select="normalize-space(/root/io/builtin:target)"/>
     <xsl:value-of select="'?/root/ctrl/state='"/>
     <xsl:value-of select="normalize-space(/root/ctrl/state)"/>
     <xsl:text disable-output-escaping="yes">&amp;</xsl:text>
     <xsl:value-of select="'/root/ctrl/action/refresh='"/>
     <xsl:text disable-output-escaping="yes">&amp;</xsl:text>
     <xsl:value-of select="'/root/ctrl/locale/country='"/>
     <xsl:value-of select="normalize-space(/root/ctrl/locale/country)"/>
     <xsl:text disable-output-escaping="yes">&amp;</xsl:text>
     <xsl:value-of select="'/root/ctrl/locale/language='"/>
     <xsl:value-of select="normalize-space(/root/ctrl/locale/language)"/>
   </xsl:template>
```

Der Refresh löst eine Transition auf einen Decision Node aus. Während der Transition wird mit einer Operation der Status des Generierungsprozesses überprüft:

```
public static boolean check()
       throws OperationException {

       ManagedMessanger mm = null;
       try {
           ServiceConnector sc = new InitialServiceConnector();
           mm = sc.getMessagingService().getManagedMessenger
(„ReportGenerator");
       }
       catch (Exception e) {
           throw new OperationException();
       }
       if (mm.messagingFinished()) {
           MessageStatus ms = mm.getStatus();
           //weiterer Ablauf abhängig vom Status
           mm.cleanUp();
           return true;
       } else {
           // zurück zur Feedback Page
           return false;
       }
   }
```

Dauert der Prozeß noch an, wird wieder zurück auf die Feedback-Seite geschaltet. Ansonsten kann auf die Ergebnisseite verzweigt werden. Eine Post Decision Operation holt den Kontoauszug aus der Datenbank. Daraufhin kann dieser angezeigt werden.

Erweiterung: Fehlerausgabe auf einer anderen Seite

Wird bei der Statusüberprüfung des asynchronen Prozesses ein Fehler festgestellt (z.B. wenn das Reporting Tool nicht antwortet, Fehler zurückliefert, ein Timeout auftritt, o.ä.), der als *OperationException* dem Benutzer signalisiert werden soll, erfolgt die Ausgabe des Fehlertextes auf dem Quell-State-Node des Decision Nodes. Dies ist in diesem Fall die Feedback-Seite. Unter Umständen ist es jedoch erwünscht, die Fehlermeldung auf einer anderen Seite auszugeben, z.B. auf derjenigen Seite, von der aus der asynchrone Prozeß gestartet wurde. Hierzu muß die Action für den Refresh als *erroraware* gekennzeichnet werden. In der Decision Operation kann dann z.B. über die Feldfunktion *hasErrors()* überprüft werden, ob Fehler aufgetreten sind. In diesem Fall wird auf den gewünschten State Node verzweigt. Dort werden die Fehlermeldungen ausgegeben.

Anmerkung

Das mitgelieferte generische Stylesheet *generic.xsl* besitzt keine Logik zur Darstellung von Refresh-Seiten.

8 Konfiguration der zen Platform

Die *zen Platform* entkoppelt die Ebene der Anwendungsentwicklung von den zugrundeliegenden komplexen J2EE-Technologien. Unabhängig vom Deployment einer *zen* Anwendung steht der Anwendungsentwicklung dadurch überall das identische, umfangreiche Instrumentarium an Services zur Verfügung. Durch die Konfiguration des *Scalable Component Framework* (SCF), der Systemebene der *zen Platform*, wird dieses an die jeweilige Laufzeitumgebung angepaßt. Die Konfiguration erfolgt getrennt für das Deployment und die Services.

Deployment

Im Rahmen der Konfiguration muß zuerst über das Deployment entschieden werden. Dabei werden verschiedene Skalierungsoptionen vorgegeben, die unter anderem die Anzahl der physikalisch beteiligten Rechnereinheiten definieren. Außerdem werden alle SCF-Components, die von der Anwendung benötigt werden, konfiguriert.

Service

Anschließend werden, jeweils abhängig von der gewählten Deployment-Variante, die Service-APIs der Anwendungsschicht mit den verfügbaren technischen Implementierungen der zugrundeliegenden Service-Spezifikationen (z.B. JDBC) gekoppelt.

Konfigurationsformat

Das Format, in dem die Konfiguration vorgenommen wird, ist weder beim Deployment noch beim Service festgelegt. Es wird beim Deployment erst durch die gewählte *DeploymentConfigurationFactory*, beim Service durch die *ServiceConnectorFactory* definiert. Daher wird im folgenden jeweils zuerst eine Referenz der Konfigurationseigenschaften dokumentiert, bevor die unterschiedlichen Konfigurationsformat beschrieben werden, in denen die Konfiguration tatsächlich vorgenommen wird.

8.1 Formate: Deployment-Konfiguration

Es ist bisher nur ein XML-Format definiert, in dem die Deployment-Konfiguration vorgenommen wird. Die DTD dazu findet sich im Anhang. Die leichten Unterschiede bzw. Anpassungen im Verhältnis zur folgenden Referenz der Deployment-Konfiguration ergeben sich intuitiv aus der DTD und den entsprechenden Konfigurationsbeispielen im Anhang. Zur Laufzeit werden die XML-Daten auf unterschiedliche Weise durch eine der *DeploymentConfigurationFactories* der *zen Platform* geladen und verwaltet.

Die Entscheidung für die adäquate DeploymentConfigurationFactory wird bei der Dokumentation der Umgebungsvariablen beschrieben.

Alle momentan verfügbaren Factories sind unterhalb des Package
*de.zeos.scf.deploy.config.**
zu finden.

8.1.1 FileBasedDeploymentConfigurationFactory

Bei Anwendung dieser *DeploymentConfigurationFactory* wird die Deployment-Konfiguration in einer XML-Datei mit Namen *scfdeploy.xml* vorgenommen, deren Struktur der Deployment-DTD entsprechen muß. Die XML-Datei muß im Klassenpfad liegen.

Da in dieser Datei das gesamte Laufzeit-Netzwerk der jeweiligen Deployment-Variante spezifiziert wird, sollte sie zentral gepflegt werden.

8.1.2 JNDIBasedDeploymentConfigurationFactory

Diese Factory erwartet einen XML-String mit der Deployment-Konfiguration im JNDI unter dem Pfad *env/zen/deploy* (namespace-freier Zugriff) der jeweiligen JVM.

8.2 Referenz: Deployment-Konfiguration

Die logische Unterteilung in Frontend und Backend beschreibt nicht unbedingt das faktische Deployment der *zen Engine*. Das Frontend kann mit dem Backend zusammen im gleichen Adreßraum laufen. Andererseits ist aber ebenso möglich, jede *zen*-Anwendung über mehrere Rechner zu verteilen, indem das Frontend auf einem eigenen Rechner und das Backend auf einem Cluster von Applikationsservern läuft. Das tatsächliche Deployment hängt nur von den Leistungsanforderungen ab, denen die Anwendung genügen muß. Ebenso ist die Zusammensetzung von Frontend und Backend in Teilen variabel. Diese Variabilität ist dadurch bedingt, daß die zentralen Module der *zen Platform* in sogenannten *SCF-Components* codiert sind. Das sind Komponenten analog zur EJB-Spezifikation von Stateless Session Beans, die im Gegensatz dazu aber von einer bestimmten Klasse der *zen Platform* erben müssen. Der Vorteil ist, daß derartige Komponenten von SCF dann auch in einer normalen Standard-JVM ausgeführt werden können.

Die zentrale physikalische Trennung zwischen Frontend und Backend erfolgt beim Aufruf der SCF-Component *Kernel*. Diese SCF-Component wird bei Cluster Deployments im Backend platziert. Sie führt die gesamte Geschäftslogik aus.

Die Geschäftslogik bleibt von den jeweils gewählten Deployment-Varianten völlig unberührt, da sie gegen die APIs der *zen Platform* programmiert ist und somit unabhängig von den zugrundeliegenden technischen Gegebenheiten auf der *zen Engine* vom jeweiligen Backend ausgeführt wird. Angepaßt wird jeweils nur die Deployment- und die Service-Konfiguration.

8.2.1 Deployment-Varianten

Die *zen Engine* kann sowohl auf einem Rechner bzw. einer *Java Virtual Machine* (JVM) oder auch verteilt auf mehreren Rechnern bzw. mehreren JVMs aufgesetzt werden. Jede beteiligte JVM-Instanz, egal ob Frontend oder Backend, wird dabei als Container bezeichnet.

➢ *Single Container Deployment*

Während der Entwicklung einer zen-Anwendung wie auch für den Fall, daß diese im produktiven Betrieb keinen unternehmenskritischen Vorgaben wie beispielsweise Ausfallsicherheit genügen muß, können Frontend und Backend auf einem Rechner laufen. Genauer gesagt werden sie in diesem Fall von der gleichen JVM im gleichen Adreßraum ausgeführt. Damit existiert bei dieser Deployment-Variante nur ein Container. Der Rechner muß dabei natürlich so ausgelegt sein, daß er die gewünschten Antwortzeiten garantieren kann.

Die vom Frontend konvertierten Anfragen werden bei diesem Deployment direkt innerhalb der JVM bzw. des Containers an das Backend weitergereicht. Die Antwort des Backends kommt ebenso direkt an das Frontend zurück.

Da alle in der *zen Platform* mitgelieferten Frontends auf die Servlet-API aufsetzen, muß in jedem Fall im Frontend auch eine Servlet-Engine in der JVM installiert sein.

➢ *Cluster Deployment*

Wenn unter anderem Ausfallsicherheit eine der Anforderungen an die Anwendung ist oder ein hohes Anfragevolumen verarbeitet werden muß, wird man die *zen Engine* normalerweise auf mehreren JVMs bzw. Rechnern verteilen. In diesem Fall werden Frontend und Backend physikalisch entkoppelt und laufen daher in verschiedenen Containern.

Jede Anfrage von einem Client wird in diesem Fall vom Frontend-Container an den Backend-Container weitergeleitet. Dieser führt die entsprechende Geschäftslogik aus und beantwortet die Anfrage. Die Antwort des Backend-Containers wird dann vom Frontend-Container wieder zurück an den Client geschickt. Die *zen Engine* im Backend läuft hier in einem J2EE-Applikationsserver.

Bei höheren Anfragevolumen wird hinter dem Frontend-Container ein Cluster von Backend-Containern platziert. Die Anfragen werden in diesem Fall vom Frontend an die verfügbaren Backends unter Beachtung der Sessionkonsistenz verteilt. Jede Session wird also bei jeder Anfrage auf exakt dem Backend bearbeitet, auf dem sie ursprünglich erstellt wurde.

Die einzelnen Backends werden bezüglich des Codes meistens identisch zusammengesetzt. Zwischen identischen Backends kann zudem die automatische Session-Replikation aktiviert werden. Es ist aber auch möglich, einzelne Aufgaben auf dedizierte Backends zu verlagern anstatt alle Backends gleichartig auszulegen.

Die einzelnen Backends müssen nicht auf identischen Applikationsservern basieren. Es können unterschiedliche Versionen des gleichen Applikationsservers genauso eingesetzt werden wie Applikationsserver unterschiedlicher Hersteller. Es bietet allerdings die naheliegenden Vorteile, nur die exakt gleiche Variante von Applikationsservern mehrfach zu replizieren.

8.2.2 Basis-Konfiguration

Ein *zen-Deployment* wird grundsätzlich unabhängig von der Deployment-Variante über die drei topologischen Einheiten *Cluster*, *Island* und *Container* konfiguriert. Außerdem werden die benötigten SCF-Components angegeben und den jeweiligen Clustern zugewiesen. Die Eigenschaften können zum großen Teil auch zur Laufzeit über die Administrationskonsole überprüft und konfiguriert werden. Die einzelnen Konfigurationsoptionen bestehen jeweils aus einem Namen und einem entsprechenden Wert.

Im folgenden werden zuerst alle von den unterschiedlichen Deployment-Varianten weitestgehend unabhängigen Konfigurationsoptionen beschrieben.

Container

Ein Container definiert die Ablaufumgebung der *zen Engine*. Bei einem Single Container Deployment existiert nur ein Container, der Frontend und Backend beinhaltet. Andernfalls ist sowohl das Frontend als auch jedes einzelne Backend ein Container. Ein Container hat die folgenden Basisattribute:

- **name**: Eine beliebige, aber eindeutige Bezeichnung für diesen Container.
- **mnemonic** (optional): Ein Kürzel für die Container-Bezeichnung. Dies wird für eine intern verkürzte Pfadcodierung bei der Identifikation des Zielcontainers einer Session und bei der Frontend-Skalierung verwendet. Ist es nicht gesetzt, wird stattdessen der Wert der der Eigenschaft *name* verwendet.

Außerdem existieren für jeden Container eine Reihe von zusätzlichen Eigenschaften:

- Konfigurationsoptionen der Initial Context Factory des Containers für den JNDI-Zugriff, je nach verwendeter Factory eventuell auch zusätzliche, hier nicht weiter ausgeführte Parameter, zumindest jedoch:
    - **java.naming.factory.initial**: Die InitialContextFactory-Klasse des Containers. (siehe auch Spezialisierungen)
- URL zur JMX-Management-Console des Containers:

- **de.zeos.scf.deploy.config.jmx.url** (optional): Die URL zur JMX-Management-Konsole (siehe auch Administration). Dies dient nur für den vereinfachten Zugriff auf alle Container-Consolen von der zentralen Management-Console aus.

Island

Ein Island faßt eine beliebige Anzahl von Containern zu einem Island zusammen. Bei einem Single Container Deployment existiert immer nur ein Island, das auch nur den einen kombinierten Container umfaßt, der zugleich Frontend und Backend ist. Bei Cluster Deployments existieren dagegen immer mindestens 2 Islands: Eines, das den Frontend-Container umfaßt und eines, über das mehrere Backend-Container zu einem Island gruppiert werden. Ein Island hat die folgenden Basisattribute:

- **name**: Eine beliebige, aber eindeutige Bezeichnung für dieses Island.

- **mnemonic** (optional): Ein Kürzel für die Island-Bezeichnung (siehe auch *mnemonic* des Containers).

Außerdem müssen alle Container aufgeführt werden, die ein Island umfaßt:

- **containerrefs**: Eine Liste aller Container, die zu diesem Island gehören. Die Liste muß disjunkt von der Menge der Container aller anderen Islands sein.

Cluster

Ein Cluster faßt mehrere Islands zu einem Cluster zusammen. Alle Container in den Islands des Clusters müssen über die gleichen SCF-Components bzw. die gleiche J2EE-Konfiguration verfügen. Dies ist für Single Container Deployments automatisch der Fall, bei Cluster Deployments müssen die einzelnen Applikationsserver geeignet konfiguriert werden. Ein Cluster hat die folgenden Basisattribute:

- **name**: Eine beliebige, aber eindeutige Bezeichnung für diesen Cluster.

- **mnemonic** (optional): Ein Kürzel für die Cluster-Bezeichnung (siehe auch *mnemonic* des Containers).

Außerdem müssen alle Islands aufgeführt werden, die ein Cluster umfaßt:

- **islandrefs**: Eine Liste aller Islands, die zu diesem Cluster gehören. Die Liste muß disjunkt von der Menge der Islands aller anderen Cluster sein.

SCF-Component

Eine SCF-Component ist die Abstraktion von einem Stateless Session Bean nach der EJB-Spezifikation. Durch die Abstraktion kann die gleiche SCF-Component sowohl in einer Standard-JVM, als auch in einem Applikationsserver genutzt werden. Die auf der technischen Schicht unterschiedlichen Aufruftechniken werden durch SCF für die Anwendungsschicht transparent gemacht. Gerade weil die zentralen Module der *zen Platform* in SCF-Components codiert sind, ergeben sich beliebige Skalierungsoptionen.

Momentan sind die folgenden Components verfügbar:

- **Kernel**: Die *zen Engine* bzw. die Kernel-Component zur Abarbeitung der Anwendungsworkflows und der Geschäftslogik. Sie muß immer definiert sein und wird logisch zum Backend gerechnet. (Klasse: *de.zeos.zen.core.comp.Kernel*).

- **XSLProcessor**: Der XSL-Prozessor wandelt die XML-Ausgabe des Backends über XSLT in das gewünschte Ausgabeformat um. Er muß ebenso wie der Kernel immer definiert sein. Logisch wird er zum Frontend gerechnet, auch wenn es Fälle geben kann, in denen er aus der Geschäftslogik heraus aufgerufen wird. (Klasse: *de.zeos.zen.ext.comp.XSLProcessor*).

- **FOPProcessor** (optional): Der FOP-Prozessor wird vom XSL-Prozessor benötigt, um PDF zu generieren. Er muß daher immer definiert sein, wenn PDF-Ausgabe benötigt wird. Er gehört normalerweise zum Frontend. (Klasse: *de.zeos.zen.ext.comp.FOPProcessor*).

- **XSDProcessor** (optional): Der XSD-Prozessor wird bei Verwendung von Web-Service-Frontends benötigt, um die Eingabedaten ins Backend-Format umzuwandeln. Diese Konvertierung erfolgt logisch gesehen im Backend (Klasse: *de.zeos.zen.ext.comp.XSDProcessor*).

Jede SCF-Component hat die folgenden konfigurierbaren Basisattribute:

- **name**: Der eindeutige Aufrufname der SCF-Component. Unter diesem Namen kann auf jede Komponente zur Laufzeit über den *ComponentSelector* zugegriffen werden. Bei den mitgelieferten Components ist das jeweils der oben angegebene Name. Bei selbst entwickelten SCF-Components ist der Name bei reinen Single Container Deployments beliebig, bei Cluster Deployments muß er aber mit dem definierten EJB-Namen aus dem entsprechenden *ejb-jar.xml* der EJB-Spezifikation übereinstimmen. Ein expliziter Zugriff auf diese Standard-Components ist bei der Anwendungsprogrammierung aber nur in sehr seltenen Spezialfällen notwendig. Die *zen Engine* selbst greift intern an den entsprechenden Stellen selbstständig über die Deployment-Konfiguration auf die notwendigen Components zu.

Außerdem existieren für jede Component eine Reihe von zusätzlichen Eigenschaften:

- **runtimeclass**: Der volle Klassenpfad der Component-Implementierung.

- **targetcluster**: Der Name des Clusters, in dem diese SCF-Component verfügbar ist. Jeder Aufruf einer Methode der entsprechenden SCF-Component wird dann aus diesen Cluster bedient. Da die *zen Engine* verteilt in verschiedenen Clustern laufen kann, muß für jede SCF-Component vorgegeben werden, in welchem der definierten Cluster respektive in welchem Set von Containern sie verfügbar ist.

8.2.3 Spezialisierung für Single Container Deployments und das Frontend bei Cluster Deployments

Container

Für Single Container Deployments werden zusätzlich folgende Attribute festgelegt:

- **remoteAccess** (default: false): Das Frontend muß als *false* definiert werden.

Außerdem müssen die folgenden Eigenschaften definiert werden:

- Eigenschaften der Initial Context Factory des Containers:
    - **java.naming.factory.initial**: Immer *tyrex.naming.MemoryContextFactory*.
    - **java.naming.factory.url.packages**: Immer *tyrex.naming*.

Island

Für Single Container Deployments werden zusätzlich folgende Attribute festgelegt:

- **failover**: Muß auf *false* gesetzt werden, da hier immer nur ein Adreßraum existiert.

SCF-Component

Zusätzlich zu den oben angegebenen Konfigurationseigenschaften können für ein Single Container Deployment für den *targetcluster* die Attribute des Component-Poolings definiert werden.

- **pooled** (optional): Mit *true* wird das Pooling aktiviert.
- **min/max** (bedingt): Wird nur gelesen, wenn *pooled=true*. Minimale und maximale Anzahl zu poolender Component-Instanzen. Die minimale Anzahl wird schon beim Start des Systems initialisiert.
- **recycle** (bedingt): Wird nur gelesen, wenn *pooled=true*. Bei *true* werden einmal angeforderte Component-Instanzen, die über die konfigurierte minimale Anzahl hinausgehen, nicht heruntergefahren, wenn sie an den Pool zurückgegeben wurden.
- **targetcluster**: Der Name des Clusters ist in diesem Fall immer der Name des einzigen definierten Clusters, da die *zen Engine* hier nicht verteilt läuft.

Ein minimales Beispiel findet sich im Anhang.

8.2.4 Spezialisierung für Backends eines Cluster Deployments

Container

Das Backend eines Cluster Deployments wird zusätzlich durch die folgenden topologischen Attribute für den Backend-Container definiert:

- **remoteAccess** (default:false): Jedes Backend muß als *true* definiert werden, da die Kommunikation zwischen Frontend und Backend über Netzwerkprotokolle erfolgt.
- **acceptingNewSessions** (default:true): Bei *true* akzeptiert dieser Container alle neuen Anfragen, bei *false* nur Folge-Anfragen von Clients, für die schon eine Session angelegt wurde, die dem Container also von vorherigen Anfragen bekannt sind. Diese Eigenschaft ist speziell für das ordnungsgemäße Herunterfahren einzelner Backends bei einem Cluster Deployment notwendig (siehe auch Administration).
- **sticky** (default:false): Ein Container, der nicht *sticky* ist, wird bei schwerwiegenden Fehlern automatisch deaktiviert. Sofern diese Eigenschaft *true* ist, bleibt der Container aber auch bei schwerwiegenden Fehlern Online. Normalerweise macht es bei Cluster Deployments Sinn, zumindest ein Backend als *sticky* zu markieren, damit das System verfügbar bleibt.

Außerdem ist eventuell die Konfiguration der folgenden Backend-Eigenschaften notwendig:

- Eigenschaften der *Initial Context Factory* des Containers:
    - Die speziellen Eigenschaften hängen vom eingesetzten Applikationsserver ab und sind dort dokumentiert.
- Zusätzliche Eigenschaften der *Initial Context Factory* des Containers:
    - **de.zeos.scf.deploy.config.comp.lookupprefix** (optional): Sofern in einem Applikationsserver für den (externen) Zugriff auf EJB-Components ein Pfad-Prefix vor dem eigentlichen EJB-Namen notwendig ist, kann hier der komplette Prefix angegeben werden.
      Damit wird verhindert, daß die Component-Namen an den Applikationsserver angepaßt werden müssen.

Island

Bei der Konfiguration des Backends eines Cluster Deployments können zusätzliche topologischen Attribute für Islands vorgegeben werden:

- **failover**: Bewirkt bei *true*, daß die automatische Session-Replikation für alle Container des Islands aktiviert wird. Sonst wird nicht repliziert. Bei Einsatz mehrerer Backends wird es durch die Replikationsmechanismen möglich, beim Ausfall eines Backends transparent und ohne Datenverlust auf ein anderes umzuschalten. Der Vorgang ist auch für den Client transparent.

- **alwaysAcceptingNewSessions** (default:false): Sofern *true* und kein Container des Islands neue Sessions akzeptiert, wird vom Island automatisch ein *sticky*-Container als *acceptingNewSessions* markiert. Existiert kein *sticky*-Container, wird ein beliebiger anderer Container selektiert.

Außerdem können die Filter-Eigenschaften überschrieben werden:

- **filter** (optional): Bei der Auswahl eines Containers zur Verarbeitung einer neuen Session selektiert der Standard-Filter *RandomContainerFilter* aus dem Package *de.zeos.scf.deploy.filter* einen Container per Zufallsalgorithmus. Neben diesem Standard-Filter existiert auch der *LoadBalancingContainerFilter*, der den Zielcontainer für neue Sessions abhängig vom Anwortverhalten der einzelnen Container verteilt. Für den Einsatz des *LoadBalancingContainerFilters* muß das *Monitoring* aktiviert sein, sonst wird automatisch auf den Standard-Filter zurückgeschaltet.

Cluster

Wie bei Islands können hier auch bei Clustern die Filter-Eigenschaften überschrieben werden:

- **filter** (optional): Bei der Auswahl eines Islands als Ziel einer neuen Anfrage selektiert der konfigurierte Cluster-Filter einen Container. Es ist normalerweise nicht nötig, den Standard-Island-Filter zu überschreiben.

Component

Für das Backend eines Cluster-Deployments gibt es eine Einschränkung bezüglich der Verteilungsmöglichkeiten einer SCF-Component. Eine Component, die nur im Frontend läuft, kann nicht von der Geschäftslogik des Backends aus aufgerufen werden.

Es macht in den meisten Fällen Sinn, den XSLProcessor in jedem Fall dem Frontend zuzuordnen, da er hier immer für die Ausgabetransformation benötigt wird. Falls der XSLProcessor bei einem Cluster-Deployment jedoch zusätzlich direkt von der Geschäftslogik benötigt wird, um Backend-Transformationen zu machen, muß er zusätzlich auch dem Backend als Stateless Session Bean zugeordnet werden. Die Component muß bei Cluster Deployments dann auch in jedem Container des entsprechenden Clusters existieren!

- **targetcluster**: Muß eine bestimmte SCF-Component sowohl im Frontend, als auch im Backend verfügbar sein, bleibt der Name des Clusters leer. Die *zen Engine* baut die Verbindung zur entsprechenden Component dann abhängig vom Aufrufcontainer als lokale Verbindung auf.
  Ansonsten steht hier der Name desjenigen Clusters, in dem die SCF-Component verfügbar ist. Das kann der Cluster sein, in dem die Geschäftslogik läuft, aber auch ein getrennt davon aufgesetzter Cluster. Für das Backend gilt, daß die entsprechende SCF-Component dann auch in allen Containern des Clusters als Stateless Session Beans verfügbar sein muß!

Verschiedene Beispielkonfigurationen finden sich im Anhang.

8.2.5 Monitoring und Journaling von Components

Das Antwortverhalten jeder Component kann zur Laufzeit automatisch überwacht werden. Dazu stehen zwei Mechanismen zur Verfügung:

Monitoring

Beim Monitoring wird die Dauer eines Request-Response-Zyklus einer Component überwacht. Beim Überschreiten vorgegebener Zeitfenster wird automatisch ein Logeintrag in den Logger mit Namen *monitoring* geschrieben. Der Logeintrag enthält neben dem Namen der Component auch das zuletzt protokollierte Antwortverhalten. Sofern der Monitoring-Logger beispielsweise mit einen Mailhandler für die Ausgabe konfiguriert ist, erhält man automatisch eine Mail, wenn die jeweilige Component unerwartet langsam antwortet. Die zusätzlichen Konfigurationsoptionen für jede Component sind:

- **alarmTimeLimit**: Mindestdauer für einen Zyklus in Millisekunden, damit das Monitoring den Zyklus protokolliert.

- **alarmThresholdIncidents**: Anzahl von Überschreitungen von *alarmTimeLimit*, um einen Alarm auszulösen, sofern diese...

- **alarmThresholdWindow**: ...innerhalb von *alarmThresholdWindow* Millisekunden auftreten.

Journaling

Das Journaling ist für die reine Protokollierung der Request-Response-Zyklen einer Component ausgelegt. Die zusätzlichen Konfigurationsoptionen für jede Component sind:

- **slots**: Anzahl von Zeitfenstern. Pro Slot werden alle Samples zusammengefaßt um eine min/max/avg-Analyse zu ermöglichen. Man kann aber auch auf alle Samples eines Slots zugreifen.
- **slotTime**: Dauer, in dem alle Samples eines Zeitfensters zusammengefaßt werden, bevor zum nächsten Slot weitergeschaltet wird. Das Zeitformat wird nach *http://www.w3.org/TR/xmlschema-2/#duration* definiert. Die Slots sind als Ringpuffer implementiert, so daß der erste Slot wieder nach der Dauer von *slot * slotTime* überschrieben wird.

Eine Beispielkonfiguration für Monitoring und Journaling findet sich im Anhang.

8.3 Formate: Service-Konfiguration

Es ist bisher nur ein XML-Format definiert, in dem die Service-Konfiguration vorgenommen wird. Die DTD dazu findet sich im Anhang. Die leichten Unterschiede bzw. Anpassungen im Verhältnis zur Referenz der Service-Konfiguration ergeben sich intuitiv aus der DTD und den entsprechenden Konfigurationsbeispielen im Anhang. Zur Laufzeit werden die XML-Daten auf unterschiedliche Weise durch eine der *ServiceConnectorFactories* der *zen Platform* geladen und verwaltet.

Die Entscheidung für die adäquate ServiceConnectorFactory wird bei der Dokumentation der Umgebungsvariablen beschrieben.

Die Service-Konfiguration existiert in den meisten Fällen für jeden Container getrennt, außer mehrere identisch konfigurierte Backends teilen sich eine Konfiguration. Alle momentan verfügbaren Factories sind unterhalb des Package
*de.zeos.scf.service.config.**
zu finden.

8.3.1 FileBasedServiceConnectorFactory

Bei Anwendung dieser *ServiceConnectorFactory* wird die Service-Konfiguration in einer XML-Datei mit Namen *scfservice.xml* vorgenommen, deren Struktur der Service-DTD entsprechen muß. In dieser XML-Datei muß die gesamte Service-Konfiguration des jeweiligen Containers spezifiziert sein. Die Datei muß im Klassenpfad liegen.

Die Service-DTD findet sich im Anhang. Die leichten Unterschiede zur relativ allgemein gehaltenen Dokumentation der Service-Konfiguration ergeben sich intuitiv aus den entsprechenden Konfigurationsbeispielen im Anhang.

8.3.2 JNDIBasedServiceConnectorFactory

Diese Factory erwartet sich einen XML-String mit der Service-Konfiguration im JNDI unter dem Pfad *env/zen/service* (namespace-freier Zugriff) der jeweiligen JVM.

8.4 Referenz: Service-Konfiguration

Neben der korrekten Deployment-Konfiguration muß auch die Konfiguration der Service-Schicht abhängig vom physikalischen Laufzeit-Container angepaßt werden. Dazu muß jeder Service auf eine Implementierung abgebildet werden, die den Service in der entsprechenden Laufzeitumgebung ausprägen kann.

Die Dokumentation der Service-Konfiguration beschreibt ausschließlich die technische Konfiguration und Adaption an die jeweiligen Laufzeit-Container eines *zen Deployments*. Der Zugriff von der Anwendungsebene auf die Service-API ist an anderer Stelle ausführlich dokumentiert.

Problematik

Im Applikationsserver ist beispielsweise der Zugriff auf JDBC-Ressourcen Bestandteil der EJB-Spezifikation. Jeder Applikationsserver stellt hierfür also eine vollständig implementierte Zugriffsschicht zur Verfügung, die der EJB-Spezifikation genügen muß, die Verbindungen normalerweise in Pools zusammenfaßt und unter anderem tote Verbindungen erkennt. Diese Schicht muß nur noch mit der Service-API der *zen Platform* verbunden werden, um der Geschäftslogik über die Service-API den Zugriff auf JDBC-Connections zu ermöglichen. Für die einfache JVM des Frontends steht der JDBC-Zugriff dagegen nicht direkt zur Verfügung, da hier keine Spezifikation existiert. Daher muß sie von der *zen Platform* selbst bereitgestellt und ebenso mit der Service-API verbunden werden.

Die Verbindung zwischen der Service-API der Anwendungsschicht und der Service-Implementierung des jeweiligen Laufzeit-Containers stellt der sogenannte *Service-Provider* her.

Service-Provider

Jeder Service, der in der Service-API definiert ist, wird über einen eigenen Service-Provider mit der Anwendungsschicht verbunden. Der Service-Provider stellt der Anwendungsebene dadurch den transparenten Zugriff auf eine adäquate Service-Implementierung aus dem jeweiligen Laufzeit-Container zur Verfügung. Sofern keine Service-Implementierung verfügbar ist, die den Bedingungen der EJB-Spezifikation genügt, muß der Service-Provider neben dem transparenten Zugriff selbst eine geeignete Implementierung bereitstellen.

Je nach Laufzeit-Container muß aus der *zen Platform* ein passender Service-Provider mit den notwendigen Eigenschaften ausgewählt und für jeden genutzten Service konfiguriert werden. Der Service-Provider eines einzelnen Services ist im Applikationsserver meistens ein anderer als bei reinen Single Container Deployments, die nur in einer Standard-JVM ablaufen.

Anders als bei der Deployment-Konfiguration, bei der nur eine zentrales Konfigurationsquelle existiert, wird für die Service-Konfiguration für jeden Container eine eigene angepaßte Konfiguration benötigt, in der die jeweils unterschiedlichen Service-Provider angegeben und konfiguriert werden. Bei einem Cluster Deployment ist es in den meisten Fällen jedoch möglich, eine einzige Service-Konfiguration für jedes Island zu teilen, da sie sich bei identischen Backends normalerweise nicht unterscheidet.

Alle Service-Provider, die Bestandteil der *zen Platform* sind, befinden sich im Package *de.zeos.scf.service.provider*.
Die vorhandenen Service-Provider decken das üblicherweise benötigte Anwendungsspektrum ab, für Spezialfälle können aber auch neue Service-Provider entwickelt werden.

Konfigurationseigenschaften

Jedem Service wird ein eigener Service-Provider zugewiesen, der den Service im entsprechenden Laufzeit-Container abbildet. Nicht benötigte Services können deaktiviert werden, indem kein Service-Provider angegeben wird. Durch den jeweils verwendeten Service-Provider werden die benötigten Konfigurationseigenschaften vorgegeben. Jede Eigenschaft besteht aus einem Namen und einem entsprechenden Konfigurationswert.

Ein Service kann über serviceglobale und poolspezifische Eigenschaften verfügen. Services, die nicht poolorientiert arbeiten, werden ausschließlich über die serviceglobalen Eigenschaften definiert. Services, die poolorientiert arbeiten, wie beispielsweise der Connection-Service, besitzen für jeden Pool zusätzlich eigene Konfigurationseigenschaften.

- **Serviceglobale Eigenschaften**:
  Die serviceglobalen Eigenschaften werden ausschließlich durch den jeweiligen Service-Provider vorgegeben.

- **Poolspezifische Eigenschaften**:
  Manche Services stellen die Möglichkeit zur Verfügung, einzelne spezialisierte Service-Instanzen zu konfigureren und über einen Namen auf diese zuzugreifen. Diese Services arbeiten poolorientiert. Jedem Service-Pool wird dabei ein eindeutiger Name zugewiesen. Die einzelnen poolspezifischen Eigenschaften, die der Service-Provider erwartet, sind für alle Pools identisch, die jeweiligen Konfigurationswerte, die die Service-Instanz definieren, sind natürlich poolspezifisch.

Im folgenden werden nun die Konfigurationseigenschaften der einzelnen Services der Service-API abhängig von den verfügbaren Service-Providern ausführlich beschrieben.

8.4.1 Logging-Service

Für den Logging-Service existiert nur ein Provider, der in allen Laufzeit-Umgebungen eingesetzt werden kann. Die Spezialisierung findet bei diesem Provider ausschließlich über seine Eigenschaften statt.
SCF setzt beim Logging auf die Bibliotheken des *Apache Jakarta Commons Logging*. Daher muß das Subsystem, auf das der Logging-Service aufsetzt, aus technischen Gründen über Umgebungsvariablen definiert werden (siehe auch dort), da es schon beim Start der jeweiligen JVM festgelegt werden muß und später nicht mehr geändert werden kann. Momentan stehen die Subsysteme der Java-Logging-API und die Log4J-API zur Verfügung.

Da in der EJB-Spezifikation kein Mechanismus zum Logging definiert wurde, ist die exakte Funktionalität abhängig von der gewählten Implementierung. Weitere Dokumentation zur Funktionalität und spezifischen Konfiguration der Logging-Subsysteme kann, soweit nicht schon im folgenden erläutert, daher nur dort nachgeschlagen werden.

➢ *LoggingServiceProvider*

Sowohl für Log4J als auch für die Java-Logging-API gilt, daß die jeweiligen Standard-Konfigurationsdateien, sofern diese verwendet werden, im Klassenpfad der *zen Engine* liegen müssen.

Bei der Konfiguration muß grundsätzlich darauf geachtet werden, welche Logging-Ausgabekanäle abhängig von der Laufzeitumgebung überhaupt eingesetzt werden können. Im Applikationsserver kann z.B. nicht in Dateien geloggt werden, weil die EJB-Spezifikation dies aus guten Gründen restriktiv handhabt. Hier bietet sich das Loggen auf die Console oder in die Datenbank an.

Log4J

Sofern als Logging-Subsystem Log4J verwendet wird, liegt die Konfiguration ausschließlich in den entsprechenden Log4J-Konfigurationsdateien, die extern gepflegt werden. Die weitere Konfiguration des LoggingServiceProviders kann in diesem Fall übersprungen und direkt bei den Erweiterungen der *zen Platform* für Log4J weitergelesen werden.

Java-Logging-API

Der Einsatz der Java-Logging-API setzt Java 1.4 voraus. Die Konfiguration kann in den jeweiligen Konfigurationsdateien der Java-Logging-API stattfinden (*logging.properties*). Die spezialisierte Konfiguration im Rahmen der *zen Platform* bietet jedoch einige Vorteile, da sie die zentrale Konfiguration der Java-Logging-API erweitert und damit umfangreiche Erweiterungen zur Verfügung stehen. Eine eventuell zusätzliche Standard-Konfigurationsdatei der Java-Logging-API wird dabei automatisch erweitert. Im folgenden wird vorausgesetzt, daß die Java-Logging-API bekannt ist.

Definition: Level-Eigenschaft

Der Name der Eigenschaft für den Level eines beliebigen Loggers definiert sich aus dem Namen des Loggers und der Endung *.level*, die an den Namen des Loggers gehängt wird. Die möglichen Werte eines Levels sind in der API zu *java.util.logging.Level* definiert.

Definition: Handler-Eigenschaft

Der Name der Eigenschaft für die Handler eines beliebigen Loggers definiert sich aus dem Namen des Loggers und der Endung *.handlers*, die an den Namen des Loggers gehängt wird. Der Wert der Eigenschaft ist jeweils der komplette Klassenname eines Handlers. Es können für jeden Logger mehrere Handler durch Komma getrennt angegeben werden. Jeder Handler muß der API gemäß *java.util.logging.Handler* genügen.

Definition: Parent

Der globale (namenlose) Logger hat keinen Parent. Ansonsten ist der Parent eines Loggers derjenige Logger, der der Vorfahre im Logging-Namespace ist (Bsp: *my.own ist* Vorfahre von *my.own.logger*).

Definition: Constructor-Eigenschaft

Normalerweise geht die Java-Logging-API davon aus, daß ein Logging-Handler einen Null-Argument-Konstruktor besitzt, der auch für die Instantiierung herangezogen wird. Sofern man einem Handler aber bei der Konstruktion zusätzliche Parameter übergeben will und dazu einen entsprechenden Konstruktor zur Verfügung stellt, kann man die einzelnen Konstruktionsparameter des Handlers als Eigenschaften in der Konfiguration angegeben. Dies kann beispielsweise dazu dienen, einem Handler einen bestimmten Formatter zuzuweisen oder Mailadressen über die Konfiguration austauschen zu können.

Der Name bzw. Parametertyp der Eigenschaft wird zusammengesetzt wie *X.construct.Y.Z*. Das Format für die einzelnen Parametertypen sieht folgendermaßen aus:

Variable	Gültige Eingabe
X	Der komplette Klassenname der Handler-Implementierung
Y	Die 1-basierte Position des Konstruktionsparameters
Z	Ein Basisdatentyp oder der komplette Klassenname einer Klasse, die als Konstruktionsparameter diesen kann. Siehe auch die folgenden Restriktionen.

Als entsprechende Konfigurationswerte sind abhängig vom oben als *Z* definierten Parametertypen sind jeweils die folgenden Eingaben gültig:

Definierter Parametertyp Z	Gültiger Parameterwert
Basisdatentyp	Wert, der von der entsprechenden Java-Wrapperklasse geparst werden kann (int, boolean, long, float, double)
Kompletter Klassenname mit 1-String-Konstruktor	Die angegebene Parameterklasse muß einen 1-Parameter-String-Konstruktor besitzen. Jeder Wert, der von dieser Klasse als String interpretiert werden kann, ist dann eine gültige Eingabe. (Beispieltyp: java.lang.String selbst).
Kompletter Klassenname mit beliebigem 1-Objekt-Konstruktor	Die angegebene Parameterklasse muß einen 1-Parameter-Konstruktor haben, auf den der als Wert angegebene komplette Klassenname anwendbar ist. Die Wert-Klasse muß *class:* als Prefix haben, zudem muß diese einen Null-Argument-Konstruktor besitzen.
Java.util.ArrayList	Ein einfacher String oder eine komma-separierte Liste von Strings, die dann automatisch in eine ArrayList umgewandelt werden.

Beispiel für einen Handler mit vorgegebenem Formatter und einer Liste von Spalten:

Konstruktor des Handlers my.FileHandler:
my.FileHandler(String s, int i, ArrayList columns, java.util.logging.Formatter f)
Konstruktor des Formatter my.Formatter:
my.Formatter()

Konfigurationsbeispiel:
my.FileHandler.construct.1.java.lang.String=/path/to/logfile
my.FileHandler.construct.2.int=48000
my.FileHandler.construct.3.java.util.ArrayList=Time, Level, Text
my.FileHandler.construct.4.java.util.logging.Formatter=class:my.Formatter

Eine ausführliche Beispielkonfiguration für den Logging-Service findet sich im Anhang.

Serviceglobale Eigenschaften

Die serviceglobalen Eigenschaften des LoggingServiceProviders definieren den globalen Logger der Java-Logging-API. Die Eigenschaftsnamen des globalen Loggers selbst beginnen jeweils mit einem Punkt ohne vorangestellten Loggernamen. Allen handlerspezifischen Einstellungen des globalen Loggers muß dagegen der komplette Klassenpfad des jeweiligen Handlers vorangestellt werden.

- **.level**: Der globale Log-Level.
- **.handlers**: Eine komma-separierte Liste der globalen Log-Handler.
- **X.level**: Der Level des globalen Handlers X. X wird in *.handlers* definiert. Sofern kein expliziter Level konfiguriert wird, ist der Parent-Level (in diesem Fall der Level des globalen Loggers) ausschlaggebend.

- **X.construct.Y.Z** (optional): Sofern spezielle Handler-Konstruktoren für definierte globale Handler verwendet werden sollen.

Poolspezifische Eigenschaften

Ein Pool des LoggingServiceProviders definiert jeweils einen einzelnen Logger. Die Eigenschaftsnamen des Loggers selbst beginnen jeweils mit einem Punkt. Allen handlerspezifischen Einstellungen des Loggers muß dagegen, wie auch bei den serviceglobalen Eigenschaften auch, der komplette Klassenpfad des jeweiligen Handlers vorangestellt werden.

- **name**: Unter dem Namen wird der Pool dem Anwendungscode über die Service-API zur Verfügung gestellt.
- **.level** (optional): Der Log-Level des Loggers. Sofern keiner angegeben ist, wird der LogLevel des Parents vererbt.
- **.handlers** (optional): Eine kommaseparierte Liste der Log-Handler dieses Loggers.
- **X.level** (optional): Der LogLevel des spezifischen Handlers X dieses Loggers. X muß in *.handlers* dieses Loggers definiert sein. Sofern kein LogLevel angegeben ist, wird der LogLevel des entsprechenden Loggers vererbt.
- **X.construct.Y.Z** (optional)**:** Sofern spezielle Handler-Konstruktoren für definierte Handler dieses Loggers verwendet werden sollen.
- **.useglobalhandlers** (default=true): Wenn true, dann erfolgt jede Logausgabe in diesen Logger zusätzlich auch auf den globalen Handlern.

➢ *Standard-Logger der zen Platform*

Die *zen Platform* benutzt intern verschiedene Logger, die in jeder Service-Konfiguration definiert werden können, um das Erkennen von Systemproblem zu vereinfachen. Sofern einer der Logger nicht definiert ist, wird der globale Logger für die Ausgabe verwendet. Dies gilt grundsätzlich, wenn ein Logger aus der Anwendungsschicht genutzt wird, der nicht explizit definiert ist.

- **system**: Die *zen Platform* gibt elementare Boot- und Core-Meldungen auf diesen Logger aus. Bestehen Probleme beim Hochfahren der Services, werden die Meldungen dieser Gruppe stattdessen über die Kommandozeile ausgegeben.
- **scf**: Das Scalable Component Framework (SCF) schreibt grundlegende technische Meldungen in diesen Logger.
- **services**: SCF gibt service-spezifische Logs über den Logger *services* aus. Dazu gehören Service-Startup- und Shutdown-Meldungen ebenso wie Fehlermeldungen aus diesem Bereich.
- **deployment**: SCF gibt deployment-spezifische Meldungen über diesen Logger aus. Das sind speziell Nachrichten, die die Erreichbarkeit von Containern oder die Verteilung von Anfragen betreffen.
- **core**: Die Kernel-Component schreibt alle Systemmeldungen, die im Zusammenhang mit der Backend-Verarbeitung einer Anfrage auftreten, in den *core*-Logger.
- **frontend**: Alle Frontend-Komponenten, speziell die Frontend-Servlets, schreiben alle Nachrichten in diesen Logger.
- **ext**: Alles, was zum Extension-System der *zen Platform* gehört, loggt nach *ext*. Das sind neben den Processor-Components auch alle Subsysteme, die den Bereich der technischen Erweiterungen (Hooks, etc) abarbeiten.
- **monitoring**: Dieser Logger wird vom Monitoring-Subsystem verwendet, um Lastprobleme bei der Überwachung einzelner Components zu melden. Da Lastprobleme meistens schwerwiegende Gründe haben, sollten die Handler dieses Loggers Ausgabeziele verwenden, die des öfteren überprüft werden, z.B. ein Handler für die Mailausgabe.

➢ *Erweiterungen der zen Platform für die Java-Logging-API*

Bei Verwendung der Java-Logging-API können ohne Einschränkung alle Handler und Formatter konfiguriert werden, die Bestandteil der Java-Logging-API sind. Die einzigen Einschränkungen bei der Anwendung betreffen die bekannten Restriktionen im Applikationsserver, die sich aus der EJB-Spezifikation ergeben.

Zusätzlich stellt die *zen Platform* die folgenden Hilfsklassen im Package *de.zeos.scf.service.log.jdk* zur Verfügung:

SimpleLineFormatter

Formatiert jeden Logeintrag, so daß die Logmeldung in einer Zeile ausgegeben wird. Nur falls Exceptions Bestandteil der Ausgabe sind, wird für jede Exception-Meldung eine weitere Zeile genutzt. Die Stacktraces von Exceptions sind nicht Bestandteil der Ausgabe.

SimpleLineExtFormatter

Formatiert jeden Logeintrag, so daß er normalerweise in einer Zeile ausgegeben werden kann. Sofern Exceptions Bestandteil der Ausgabe sind, werden mehrere Zeilen genutzt. Auch die Stacktraces der Exceptions werden von diesem Formatter ausgegeben.

ExtConsoleHandler

Der *ExtConsoleHandler* erweitert den java.util.logging.ConsoleHandler um einen Konstruktor zur Angabe eines beliebigen Formatters direkt in der Konfiguration.

EnhancedFileHandler

Der *EnhancedFileHandler* erweitert den *java.util.logging.FileHandler* um einen Konstruktor mit mehreren Parametern, die dadurch direkt in der Konfiguration eingestellt werden können:

- **pattern**: Ein Pattern analog zur Logging-API, um den Dateinamen für Rollover einzustellen. Der Pfad zur Logdatei muß existieren!
- **limit**: Die maximale Größe einer Logdatei in Bytes.
- **count**: Die Anzahl von Dateien, die der Handler für Rollover nutzen kann (Ringpuffer).
- **append**: Bei *true* werden neue Logeinträge an vorhandene Dateien angehängt, bei *false* werden vorhandene Rollover-Dateien beibehalten, nur die 0-Datei wird neu angelegt. Bei Verwendung des XML-Formatters kann es hier Probleme geben (siehe auch Java-Bug-ID 4629315).
- **formatter**: Eine Formatter-Klasse für das Ausgabeformat, z.B. der SimpleLineFormatter.

JdkMessagingDBHandler

Der JdkMessagingDBHandler schreibt Logeinträge in eine Datenbank. Die Logeinträge werden zunächst gepuffert, bevor sie in eine Message Queue gestellt werden, wo sie ein Message Bean abholt und anschließend in die Datenbank schreibt.

Der Zugriff auf die Datenbank erfolgt über einen Pool des *ConnectionService*, der *zenlog* heißen muß. In der entsprechenden Datenbank muß eine Tabelle *zenlog* existieren, die folgender DDL genügt:

```
CREATE TABLE ZENLOG (
  POT        DATE            NOT NULL,
  LOGNAME    VARCHAR2 (30),
  LOGLEVEL   VARCHAR2 (10)   NOT NULL,
  MESSAGE    VARCHAR2 (4000) NOT NULL,
  CONTAINER  VARCHAR2 (50),
  SEQUENCE   NUMBER (19)  );
```

Außerdem muß ein *MessagingHandler* mit dem festgelegten Namen *DBMessageLogger* (siehe auch dort) konfiguriert sein, der auf die Message-Bean-Implementierung *de.zeos.scf.service.log.DBMessageLogger* abgebildet ist. Dieses Message Bean holt die Logeinträge aus der Queue und schreibt sie gesammelt in die Datenbank.

Die Konstruktoren für den JdkMessagingDBHandler sind:

- **()**: Default-Konstruktor mit Puffergröße 5.
- **(int size)**: Konstruktor mit konfigurierbarer Puffergröße.

JdkMessagingMailHandler

Der *JdkMessagingMailHandler* sendet Logeinträge an eine oder mehrere E-Mail-Adressen. Der Einsatz dieses Handlers ist sehr vorteilhaft, wenn man ihn zusätzlich neben dem üblichen Dateilog einsetzt. Der Dateilogger sichert dann je nach Log-Level fast alle Logeinträge, dem MailHandler wird ein hoher Log-Level zugewiesen, so daß er nur schwerwiegende Fehler oder Warnungen meldet. Durch die sofortige Meldung eines schwerwiegenden Problems per Mail kann deutlicher schneller reagiert werden, als wenn nur ab und zu die Logdateien kontrolliert werden.

Die Logeinträge werden nicht direkt von diesem Mail Handler versandt, sondern in eine Message Queue gestellt, wo sie ein Message Bean abholt und anschließend an die konfigurierten Empfänger versendet. Daher muß ein *MessagingHandler* mit dem festgelegten Namen Mail*MessageLogger* (siehe auch dort) konfiguriert sein, der auf die Message-Bean-Implementierung *de.zeos.scf.service.log.MailMessageLogger* abgebildet ist. Dieses Message Bean holt die Logeinträge aus der Queue und schließt den Mail-Versand ab.

Das Versenden eines Logeintrages per Mail ist allerdings deutlich ressourcenaufwendiger als beispielsweise der Eintrag in einer Log-Datei oder die gepufferte Ablage in der Datenbank. Daher müssen Vorkehrungen getroffen werden, um potentielle Mailstorms schon an der Quelle abzufangen.

Ein Mailstorm kann allgemein als massives unkontrollierbares Auftreten von Mailversand bzw. -empfang verstanden werden. Dieses Phänomen kann beispielsweise entstehen, wenn unerwartete Fehler oder Zustände in der Geschäftslogik auftauchen, und diese durch Logeinträge über einen MailLogger kommuniziert werden. Schlimmstenfalls tritt das Problem sogar in einer Programmschleife auf, die bei jeder Iteration neue Logeinträge generiert. Zudem kann es vorkommen, daß der fehlerhafte Code je nach Anwendungsfall mit jeder neuen Anfrage erneut ausgeführt wird. Sofern der fehlerhafte Code zu einer Fehlerseite führt, führen die Benutzer zudem erfahrungsgemäß die gleiche Anfrage immer wieder aus, um eine korrekte/erwartete Antwort bzw. Ausgabeseite des Systems zu bekommen.
Die Zahl der indirekt durch Logeinträge generierten Mails kann sich dadurch je nach Anfragevolumen zu einem Mailstorm hochschaukeln.

Derartige Problemfälle lassen sich im Vorfeld meistens nur schwierig ausreichend abschätzen und testen. Um die Mailbox der Empfänger in so einem Fall nicht durch massenweise identische Mails zu blockieren, läuft der *JdkMessagingMailHandler* mit einer integrierten

automatischen Mailstorm-Erkennung. Mails mit identischem Inhalt werden so vor dem Versand gesammelt; der Inhalt wird nur einmal mit einer Zusammenfassung verschickt. Ebenso wird auch ein massenhaftes Auftreten verschiedenartiger Mails automatisch erkannt und diese zu einer Mail gebündelt. Damit bleibt sichergestellt, daß das System auch bei schwerwiegenden Problemen weiter reaktiv bleibt.

Der Algorithmus für die Mailstorm-Erkennung kann optional durch verschiedene Parameter an die spezifischen Bedürfnisse angepaßt werden. Für die meisten Standardfälle ist allerdings der Standard-Konstruktor ausreichend. Die beiden verfügbaren Konstruktoren sind:

- **(ArrayList recipientList, String sender)**
  - **recipientList**: Es können ein oder mehrere Empfänger als Komma-separierte Empfängerliste für diesen Handler angegeben werden.
  - **sender**: Als Absender der Log-Mails wird *sender* eingetragen. Dies ist keine E-Mail-Adresse, sondern ein beliebiger kurzer Begriff, der den Absender kenntlich machen soll. Bevor die Startup-Phase der zen Engine abgeschlossen ist, wird als tatsächlicher Absender *sender@deployment-initing* eingetragen. Nach dem Systemstart wird jeder Logeintrag dann als *sender@containername* zugestellt, um bei verteilten Anwendungen den Container identifizieren zu können, der der Auslöser für den Logeintrag war.
- **(ArrayList recipientList, String sender, int identicalPerUnitTrigger, int identicalMaxUnitMillis, int identicalMailStormFlushCount, int identicalMailStormUnitFlushCount, float identicalVirtualMailStormCacheExtension, int singularPerUnitTrigger, int singularMaxUnitMillis, int singularMailStormFlushCount, float singularVirtualMailStormCacheExtension, int silenceMillisAfterMailStormForAutoFlush, int singularMailContentLinesExtraction**
  - **recipientList**: wie oben
  - **sender**: wie oben
  - **identicalPerUnitTrigger**: Anzahl identischer Logeinträge, die das Mailstorm-Handling auslösen, sofern diese innerhalb von...
  - **identicalMaxUnitMillis**: ...Millisekunden auftreten. Alle Logeinträge zuvor werden direkt verschickt.
  - **identicalMailStormFlushCount**: Maximale Anzahl jeweils identischer Logeinträge, die nach Beginn der Mailstorm-Erkennung in einer Unit zwischengespeichert werden. Ist diese Zahl für eine beliebige Unit erreicht, wird der Logeintrag der Unit mit einer Zusammenfassung verschickt.
  - **identicalMailStormUnitFlushCount**: Maximale Anzahl erkannter Mailstorms bzw. vorhandener Units. Ist diese Zahl erreicht, werden alle Units getrennt (aber jeweils zusammengefaßt) verschickt, auch wenn einzelne Units noch nicht den Wert *identicalMailStormFlushCount* erreicht haben.
  - **identicalVirtualMailStormCacheExtension**: Faktor, um den die maximale Anzahl gesammelter Units virtuell erweitert wird, bevor der Versand erfolgt. Dies muß exakt *1.0f* sein, wenn keine einzige Unit verloren gehen darf, ansonsten ein Wert *>1.0f*. Ein größerer Wert reduziert die Serverlast, indem einige Units verloren gehen, allerdings werden die einzelnen Logeinträge normalerweise auch noch in Datei- oder Datenbanklogger geschrieben und können dort überprüft werden.
  - **singularPerUnitTrigger**: Anzahl von Unikat-Logeinträgen, die das Mailstorm-Handling auslösen, sofern diese innerhalb von...
  - **singularMaxUnitMillis**: ...Millisekunden auftreten. Alle Logeinträge zuvor werden direkt verschickt.
  - **singularMailStormFlushCount**: Maximale Anzahl von Unikat-Logeinträgen, die nach Beginn der Mailstorm-Erkennung zwischengespeichert werden. Ist diese Zahl erreicht, werden alle einzelnen Unikat-Logeinträge zusammengefaßt und als eine Mail verschickt.
  - **singularVirtualMailStormCacheExtension**: Faktor, um den die maximale Anzahl gesammelter Unikat-Logeinträge virtuell erweitert wird, bevor der Versand erfolgt. Dies muß exakt *1.0f* sein, wenn kein Logeintrag verloren gehen darf, ansonsten ein Wert *>1.0f*.
  - **silenceMillisAfterMailStormForAutoFlush**: Dauer in Millisekunden ohne Mailstorm, bis alle verbliebenen Logeinträge verschickt werden. Die Dauer wird mit jedem neuen eingehenden Logeintrag überprüft.
  - **singularMailContentLinesExtraction**: Anzahl der Zeilen, die von jedem Unikat-Logeintrag zwischengespeichert und bei Erreichen eines Triggers oder nach einem Mailstorm als Zusammenfassung verschickt werden.

➢ *Erweiterungen der zen Platform für Log4J*

Log4jMessagingDBAppender

Der *Log4JMessagingDBAppender* hat die gleichen Eigenschaften wie die Version für die Java-Logging-API. Momentan kann man den Appender nur durch Vererbung und das Überschreiben der Konstruktoren konfigurieren.

Log4JMessagingMailAppender

Der *Log4JMessagingMailAppender* hat die gleichen Eigenschaften wie die Version für die Java-Logging-API. Momentan kann man den Appender nur durch Vererbung und das Überschreiben der Konstruktoren konfigurieren.

8.4.2 Mail-Service

Der Mail-Service stützt sich analog zur EJB-Spezifikation auf die Java-Mail-API. Zur Anbindung des Mail-Service existieren abhängig von der Laufzeitumgebung verschiedene Provider. Die Konfigurationsoptionen der Provider sind trotzdem ähnlich. Sie beschränken sich auf serviceglobale Eigenschaften, das heißt auch, daß momentan pro Container nur ein Mailserver angesteuert werden kann.

➢ *MailServiceJNDIProvider*

Der *MailServiceJNDIProvider* wird in Containern eingesetzt, bei denen keine Mail-Session über den JNDI des gleichen Adreßraums verfügbar ist. Dies ist im allgemeinen der Fall für Single Container Deployments und für das Frontend bei Cluster Deployments.

Serviceglobale Eigenschaften

- **jndiaccess** (optional): Der Kontextpfad, unter dem der Provider eine Mail-Factory vom Typ *javax.mail.Session* im JNDI verfügbar macht. Fehlt dieser Parameter (->Standardfall), wird der EJB-bzw. Java-Mail-API-Standard *comp/env/mail* verwendet.
- **jndisessionname**: Der Name, unter dem dieser Provider die Mail-Factory unter dem Kontextpfad *jndiaccess* im JNDI bindet.

Die Konfiguration des Mailserver-Zugriffs selbst liegt, analog zur Java-Mail-API, in der Datei *mail.properties*, die im Klassenpfad verfügbar sein muß.

➢ *PreloadedMailServiceJNDIProvider*

Die Backends bei Cluster Deployments haben nach EJB-Spezifikation JNDI-Zugriff auf eine Mail-Session, sofern ein Mailserver verfügbar ist. Daher ist hier der *PreloadedMailServiceJNDIProvider* ausreichend, der über JNDI nur eine Verbindung zur vorkonfigurierten Mail-Session herstellt. Die exakten Eigenschaften der Mail-Session werden hier in der Konfiguration des Applikationsservers vorgegeben.

Serviceglobale Eigenschaften

- **jndiaccess** (optional): Der Kontextpfad, unter dem der EJB-Deployer die Mail-Factory vom Typ *javax.mail.Session* im JNDI abgelegt hat. Fehlt dieser Parameter (->Standardfall), wird der EJB-bzw. Java-Mail-API-Standard *comp/env/mail* verwendet.
- **jndisessionname**: Der Name, unter dem die Mail-Session vom EJB-Deployer unter dem Kontextpfad *jndiaccess* im JNDI gebunden wurde.

Die Konfiguration des Mailserver-Zugriffs selbst muß hier entsprechend der Dokumentation des eingesetzten Applikationsservers vorgenommen werden.

8.4.3 Transaction-Service

Der Transaction-Service stützt sich auf die Java-Transaction-API. Er stellt den Zugriff auf den Transaktionsmanager des Systems und entsprechende *UserTransactions* bereit. Zur Anbindung des Transaction-Service existieren abhängig von der Laufzeitumgebung verschiedene Provider. Die Konfigurationseigenschaften der Provider beschränken sich auf serviceglobale Eigenschaften.

➢ *TransactionServiceJNDIProvider*

Der *TransactionServiceJNDIProvider* wird in Containern eingesetzt, bei denen kein Transaktionsmanager per Spezifikation verfügbar ist. Dies ist im allgemeinen der Fall für Single Container Deployments und für das Frontend bei Cluster Deployments.

Serviceglobale Eigenschaften

- **jndiaccess** (optional): Der Kontextpfad, unter dem der Provider den Transaktionsmanager und die *UserTransaction* im JNDI verfügbar macht. Fehlt dieser Parameter (->Standardfall), wird für beides der EJB-Standard für *UserTransactions* der Pfad *comp* verwendet.
- **tmjndiinstancename**: Der Name, unter dem der Transaktionsmanager vom Typ *javax.transaction.TransactionManager* unter dem Kontextpfad *jndiaccess* im JNDI gebunden wird.
- **utxjndiinstancename**: Der Name, unter dem die *UserTransaction* vom Typ *javax.transaction.UserTransaction* unter *jndiaccess* im JNDI gebunden wird.

- **timeout** (default=60): Die maximale Dauer einer Transaktion in Sekunden, bevor automatisch ein Rollback erfolgt.
- **maxtransactions** (default=50): Maximale Anzahl von gleichzeitig geöffneten Transaktionen.

➢ *PreloadedTransactionServiceJNDIProvider*

Die Backends bei Cluster Deployments haben nach EJB-Spezifikation automatisch Zugriff Transaktionsobjekte vom Typ *javax.transaction.UserTransaction*. Daher ist hier der *PreloadedTransactionServiceJNDIProvider* ausreichend, der über JNDI nur eine Verbindung zur vorkonfigurierten *UserTransaction* herstellt. Die Eigenschaften des Transaktionsmanagers werden hier in der Konfiguration des Applikationsservers eingestellt. Je nach Applikationsserver kann es sein, daß der Zugriff auf den Transaktionsmanager selbst nicht möglich ist.

Serviceglobale Eigenschaften

- **utxjndiinstancename**: Der Name, unter dem die *UserTransaction* vom EJB-Deployer unter dem Kontextpfad *comp* im JNDI gebunden wurde.

Die Konfiguration des Transaktionsmanagers selbst muß hier entsprechend der Dokumentation des eingesetzten Applikationsservers vorgenommen werden.

8.4.4 Connection-Service

Der Connection-Service stützt sich analog zur EJB-Spezifikation auf die JDBC-API. Zur Anbindung des Connection-Service existieren abhängig von der Laufzeitumgebung verschiedene Provider.

➢ *ConnectionServiceJNDIProvider*

Der *ConnectionServiceJNDIProvider* wird in Containern eingesetzt, bei denen keine Connection-Factory vom Typ *javax.sql.DataSource* über den JNDI des gleichen Adreßraums verfügbar ist. Dies ist im allgemeinen der Fall für Single Container Deployments und für das Frontend bei Cluster Deployments.

Jedem definierten Pool kann eine bestimmte Anzahl von Datenbankverbindungen fest zugeordnet werden. Ein Pool kann allerdings auch *virtuell* konfiguriert werden, solange er keinen eigenen physikalischen Pool benötigt. Er nutzt dann die Verbindungen eines anderen, adäquaten Pools mit. Dadurch kann man dem Anwendungscode von Beginn eine, logisch gesehen, optimale Einteilung von Connectionpools zur Verfügung stellen, muß aber nicht für jeden Pool eigene physikalische Verbindungen aufbauen. Jeder virtuelle Pool kann jederzeit per Konfiguration in einen echten physikalischen Connectionpool umgewandelt werden, ohne den Anwendungscode zu tangieren.

Serviceglobale Eigenschaften

- **jndiaccess** (optional): Der Kontextpfad, unter dem der Provider eine Connection-Factory vom Typ *javax.sql.DataSource* im JNDI verfügbar macht. Fehlt dieser Parameter (->Standardfall), wird der EJB-bzw. JDBC-API-Standard *comp/env/jdbc* verwendet.
- **preload**: Eine kommaseparierte Liste von Standard-Datenbank-Treiberklassen, die vor dem Aufbau der Connectionpools instanziiert werden müssen. Transaktionale Datenbanktreiber können Bestandteil der Liste sein, wenn sie für Standard-Connections verwendet werden sollen. Bei Definition eines Connectionpools als transaktionale Ressource muß der jeweilige transaktionale Treiber dagegen zusätzlich explizit zusammen mit dem jeweiligen Pool konfiguriert werden.

Poolspezifische Eigenschaften

- **name**: Der Name, unter dem der Connectionpool unter dem Kontextpfad *jndiaccess* im JNDI gebunden werden soll. Unter diesem Namen wird der Pool dem Anwendungscode über die Service-API zur Verfügung gestellt.
- **jndiinstancename**: Der Name des Connectionpools, aus dem dieser logische Pool seine physikalischen Datenbank-Connections bezieht. Sofern der Name identisch mit dem *name* dieses Pools ist, wird ein physikalischer Pool definiert, für den weitere Optionen konfiguriert werden müssen.
  Um einen virtuellen Pool zu definieren, wird stattdessen als Wert der Name eines physikalischen Pools angeben, der verwendet werden soll. In diesem Fall differieren Poolname und *jndiinstancename*. Alle weiteren Eigenschaften werden dann nicht mehr benötigt.
- **transactional** (optional): Sofern der Connectionpool als transaktionale Ressource verwaltet werden soll, muß der Wert hier *true* sein. Die Connection-Factory wird dann als *XAResource* initialisiert und dem Transaktionsmanager zugeordnet. Der passende Datenbanktreiber für diesen Pool muß dann explizit als *xadriver* gesetzt sein.
  Ist der Wert *false* oder fehlt diese Option, wird der Connectionpool als Standardpool hochgefahren und automatisch der korrekte Treiber aus den unter *preload* konfigurierten Treibern ausgewählt.
- **xadriver** (bedingt): Sofern der Connectionpool als transaktional gekennzeichnet ist, muß hier der passende optimierte XA-Treiber für die Zieldatenbank angegeben werden. Da es keinen Standard für die Datenübergabe an einen XA-Treiber gibt, muß der Treiber indirekt durch einen passenden Wrapper angeprochen werden. Daher wird nicht der Treiber direkt, sondern der volle Klassenname des jeweiligen Wrappers

als Wert für *xadriver* angegeben.
Momentan existiert der spezialisierte Wrapper
de.zeos.scf.core.db.XADataSourceOracleImpl
für Oracle-Datenbanktreiber. Liegen keine datenbankspezifisch optimierten XA-Treiber vor, kann der Wrapper
de.zeos.scf.core.db.XADataSourceMinervaImpl
eingesetzt werden, der auf Basis des passenden JDBC-Treibers, der unter *preload* konfigurert sein muß, einen XA-Treiber simuliert.

- **url**: Die volle Connection-URL, die den Datenbankzugriff definiert.
- **user**: Der Benutzername für die Datenbankverbindung.
- **password**: Das Paßwort für die Datenbankverbindung.
- **min**: Minimale Anzahl von bereitgehaltenen Connections.
- **max**: Maximale Anzahl von bereitgehaltenen Connections.
- **blocking** (default=true): Steuert das Poolverhalten, wenn gerade keine Connection frei ist. Bei *true* muß der Client, der die Connection anfordert, warten bis eine freigegeben wird. Bei *false* wird sofort *null* zurückgegeben; der Client muß darauf geeignet reagieren, kann es aber auch sofort erneut probieren.
- **blocking-timeout** (default=-1): Maximale Dauer in Millisekunden, die bei *blocking=true* auf eine freie Verbindung gewartet wird. Bei -1 wird so lange gewartet, bis eine Verbindung frei ist.

➢ *PreloadedConnectionServiceJNDIProvider*

Die Backends bei Cluster Deployments haben nach EJB-Spezifikation JDBC-Zugriff. Daher ist hier der *PreloadedConnectionServiceJNDIProvider* ausreichend, der über JNDI nur eine Verbindung zum vorkonfigurierten Connectionpool herstellt. Die exakte Optionen der einzelnen Pools werden hier in der Konfiguration des Applikationsservers eingestellt.

Serviceglobale Eigenschaften

- **jndiaccess** (optional): Der Kontextpfad, unter dem die einzelnen Connectionpools vom EJB-Deployer unter ihrem Poolnamen im JNDI angelegt wurden. Fehlt dieser Parameter (->Standardfall), wird der EJB-bzw. JDBC-API-Standard *comp/env/jdbc* verwendet.

Poolspezifische Eigenschaften

- **name**: Der Name des Pools, unter dem er dem Anwendungscode über die Service-API zur Verfügung gestellt wird.
- **jndiinstancename**: Der Name des physikalischen Pools aus dem JNDI, auf den dieser logische Pool gemappt werden soll. Da bei diesem Service-Provider kein Pool explizit definiert und physikalisch hochgefahren wird, ist quasi jeder Pool ein virtueller Pool. Die physikalische Konfiguration findet im entsprechenden Container/Applikationsserver statt.

8.4.5 JDO-Service

Der JDO-Service stellt den Zugriff auf die Persistenzschicht über die *Java Data Objects* Spezifikation zur Verfügung. Die *zen Platform* nutzt den JDO-Service, um auf das Repository der einzelnen *zen* Anwendungen zuzugreifen.

Zur Anbindung des JDO-Service existieren abhängig von der Laufzeitumgebung verschiedene Provider. Momentan stützen sich alle Provider auf die JDO-Implementierung *Kodo* der Firma S*olarMetric*. JDO ist momentan noch nicht in der EJB-Spezifikation definiert.

Jedem JDO-Pool muß ein Connectionpool zugewiesen werden, über den der Zugriff auf die entsprechende Datenbank erfolgen kann. Der Connectionpool muß nicht explizit für den JDO-Pool reserviert sein, er kann gleichzeitig auch direkt über den ConnectionService oder durch andere JDO-Pools genutzt werden.

➢ *JDOServiceJNDIProvider*

Der *JDOServiceJNDIProvider* wird in Containern eingesetzt, bei denen keine *PersistenceManagerFactory* vom Typ *javax.sql.DataSource* über den JNDI des gleichen Adreßraums verfügbar ist. Da die JDO-API momentan noch nicht Bestandteil der EJB-Spezifikation ist, existieren bisher nur wenige Container, die JDO-Factories direkt zur Verfügung stellen können.

Serviceglobale Eigenschaften

- **jndiaccess** (optional): Der Kontextpfad, unter dem der Provider eine *PersistenceManagerFactory* vom Typ *javax.jdo.PersistenceManagerFactory* im JNDI verfügbar macht. Fehlt dieser Parameter (->Standardfall), wird der JDO-API-Standard *comp/env/jdo* verwendet.

Poolspezifische Eigenschaften

- **name**: Der Name, unter dem die *PersistenceManagerFactory* für diesen Pool unter dem Kontextpfad *jndiaccess* im JNDI gebunden werden soll. Auf den Pool kann im Anwendungscode namensbasiert über die Service-API zugegriffen werden.

- **connectionservice**: Der Name des Connectionpools, den dieser JDO-Pool für den Datenbankzugriff verwenden soll. Dieser Pool muß in der *ConnectionService*-Konfiguration existieren; es sind auch virtuelle Pools erlaubt.

➢ *PreloadedJDOServiceJNDIProvider*

Sofern ein Container eine *PersistenceManagerFactory* im JNDI zur Verfügung stellt, ist der *PreloadedConnectionServiceJNDIProvider* ausreichend, der über JNDI nur eine Verbindung zu den vorkonfigurierten Instanzen der *PersistenceManagerFactory* herstellt. Die exakte Optionen der einzelnen JDO-Pools werden dann in der Konfiguration des Laufzeit-Containers/Applikationsservers definiert.

Die Eigenschaften sind identisch zu denen des *JDOServiceJNDIProviders*, allerdings wird nur lesend auf sie zugegriffen. Der JNDI wird daher auch nicht modifiziert.

➢ *JDOServiceInMemoryProvider*

Sofern man in einem Container ohne JDO-Unterstützung keinen schreibenden Zugriff auf den JNDI hat, kann man auch den *JDOServiceJNDIProvider* nicht anwenden, um die JDO-Factories nachzuladen. Der *JDOServiceInMemoryProvider* bietet hier Abhilfe, da er nach dem Laden über die Service-API den direkten Zugriff auf die JDO-Objekte innerhalb des Adreßraums und ohne JNDI möglich macht.

Die Eigenschaften dieses Providers sind identisch zu denen des *JDOServiceJNDIProviders*.

8.4.6 Messaging-Service

Für den Messaging-Service existiert als Service-Provider nur der *MessagingServiceProvider*. Bisher sind keine serviceglobalen Einstellungen spezifiziert, die Konfiguration erfolgt daher nur über poolspezifische Eigenschaften. Die Spezialisierung der einzelnen Message-Pools erfolgt abhängig vom Laufzeit-Container über entsprechende *MessagingHandler* und deren jeweilige Eigenschaften. Alle *MessagingHandler* der *zen Platform* befinden sich im Package *de.zeos.scf.service.msg*.

Da der Messaging-Service analog zu Java Message Beans arbeitet, muß für jeden Message-Pool als *Message Consumer* auch eine Implementierung eines Message Beans vorhanden sein, das die jeweiligen Nachrichten empfängt und verarbeitet.

➢ *Basis-Konfiguration*

Die Basiskonfiguration eines Messaging-Pools ist für alle *MessagingHandler* identisch.

Poolspezifische Eigenschaften

- **name**: Der Name, unter dem man aus dem Anwendungscode heraus über die Service-API auf den der Messenger des Pools zugreifen kann.
- **handler**: Hier muß der volle Klassenname des adäquaten *MessagingHandlers* gesetzt sein (siehe Spezialisierungen).
- **timeout**: Wird nur für *ManagedMessenger* benötigt. Definiert die Dauer, die einem *ManagedMessenger* für die Verarbeitung eingeräumt wird, bevor eine *ManagedMessage* als Timeout deklariert wird.

➢ *MessengerLocalImpl*

Dieser *MessagingHandler* wird in Containern eingesetzt, bei denen keine JMS-*ConnectionFactory* vom Typ *javax.jms.QueueConnectionFactory* über den JNDI des gleichen Adreßraums verfügbar ist. Dies ist im allgemeinen der Fall für Single Container Deployments und für das Frontend bei Cluster Deployments. Die *MessengerLocalImpl* simuliert die notwendigen Module einer echten Implementierung des Java Message Service. Da die Implementierung auf Threads aufsetzt, darf sie nicht im Applikationsserver eingesetzt werden.

Neben den poolspezifischen Eigenschaften aus der Basis-Konfiguration wird dieser *MessagingHandler* durch die folgenden Optionen definiert:

Poolspezifische Eigenschaften

- **handler**: *de.zeos.scf.service.msg.MessengerLocalImpl*.
- **queueName**: Ein eindeutiger virtueller Name für diesen *MessagingHandler*.
- **systemName**: Der volle Klassenname des Message Consumer Beans für die Verarbeitung der Nachrichten. Abhängig von der Verwendung als Messenger oder *ManagedMessenger* muß diese Klasse die notwendige Hierarchie ausprägen.

➢ *MessengerJMSImpl*

Die Backends bei Cluster Deployments haben nach EJB-Spezifikation JMS-Zugriff. Daher kann hier die *MessengerJMSImpl* eingesetzt werden, der über JNDI nur eine Verbindung zur vorkonfigurierten JMS-*ConnectionFactory* herstellt.

Die entsprechenden JMS-Queues, an die die Nachrichten der einzelnen Pools zugestellt werden, müssen hier in der Konfiguration des Applikationsservers konfiguriert werden. Der *MessagingHandler* reicht die eingehenden Nachrichten nur an die jeweiligen Queues weiter.

Neben den poolspezifischen Eigenschaften aus der Basis-Konfiguration wird dieser *MessagingHandler* durch die folgenden Optionen definiert:

Poolspezifische Eigenschaften

- **handler**: *de.zeos.scf.service.msg.MessengerJMSImpl.*
- **connectionFactory**: Der vollständige Pfad, unter dem die JMS-*ConnectionFactory* im JNDI abgelegt ist, inklusive des eventuell notwendigen Namespaces und des Namens der *ConnectionFactory*, *z.B.*: *MyDirectConnectionFactory* oder *java:comp/env/jms/MyConnectionFactory*.
- **queueName**: Der Name der konfigurierten JMS-Queue, an die die Nachrichten zugestellt werden sollen. Das entsprechende Message Consumer Bean für die Abarbeitung der Nachricht muß im jeweiligen Laufzeit-Container/Applikationsserver als Empfänger für diese Queue konfiguriert sein.
- **systemName**: Der volle Klassenname des Message Consumer Beans für die Verarbeitung der Nachrichten.

8.4.7 ResourceRepository-Service

Für den *ResourceRepository*-Service existiert als Service-Provider nur der *ResourceRepositoryServiceProvider*. Bisher sind keine serviceglobalen Einstellungen spezifiziert, die Konfiguration erfolgt daher nur über poolspezifische Eigenschaften. Die Spezialisierung der einzelnen *ResourceRepository*-Pools erfolgt abhängig vom Laufzeit-Container und dem jeweiligen Einsatzzweck über entsprechende *ResourceRepository*-Handler und deren Eigenschaften. Alle Handler der *zen Platform* befinden sich im Package *de.zeos.scf.service.res*.

➢ *Basis-Konfiguration*

Die Basiskonfiguration eines *ResourceRepository*-Pools ist für alle Handler identisch.

Poolspezifische Eigenschaften

- **name**: Der Name, unter dem man aus dem Anwendungscode heraus über die Service-API auf dieses *ResourceRepository* zugreifen kann.

➢ *FileResourceRepositoryHandler*

Dieser *ResourceRepositoryHandler* verwaltet beliebige Daten auf dem Dateisystem. Die Möglichkeit zum Zugriff auf das Dateisystem ist daher eine Vorbedingung, um diesen Handler einsetzen zu können. Da dies nach der EJB-Spezifikation verboten ist, kann dieser Handler nicht im Applikationsserver eingesetzt werden.

Der Zugriff ist unbeschränkt, das heißt man kann über diesen Handler Dateien lesen, schreiben und löschen. Neben den poolspezifischen Eigenschaften aus der Basis-Konfiguration wird dieser Handler durch die folgenden Optionen definiert:

Poolspezifische Eigenschaften

- **url**: Eine gültige URL nach dem *file*-Protokoll, die auf das (Unter-)Verzeichnis auf dem Dateisystem zeigt, unter dem dieses *ResourceRepository* arbeiten soll. Das Verzeichnis wird vom Handler dann als Wurzelverzeichnis dieses *ResourceRepository* freigegeben. Ob eine URL gültig ist, hängt nicht zuletzt auch vom Betriebssystem ab. Beispiele:
 file:c:/myroot
 file:///c:/myroot
 file:/opt/myroot

➢ *HttpResourceRepositoryHandler*

Dieser *ResourceRepositoryHandler* stellt den Ressourcen-Zugriff über einen Http-Server zur Verfügung. Vorbedingung für die Anwendung dieses Handlers ist also, daß ein geeigneter Http-Server bereitsteht, von dem dieser Handler per Http-Protokoll die jeweiligen Daten lesen und, je nach Http-Server-Implementierung, eventuell auch schreiben kann. Da die Verwendung des Http-Protokolls auch nach der EJB-Spezifikation erlaubt ist, kann dieser Handler in jedem Container eingesetzt werden. Der Http-Server kann auch durch Definition eines *ResourceRepository* im Frontend über den *JettyHttpResourceRepositoryHandler* gestartet werden (siehe unten).

Neben den poolspezifischen Eigenschaften aus der Basis-Konfiguration wird dieser Handler durch die folgenden Optionen definiert:

Poolspezifische Eigenschaften

- **url**: Die vollständige Http-URL auf das jeweilige (Unter-)Verzeichnis eines laufenden Http-Servers, unter dem dieses *ResourceRepository* arbeiten soll. Das Verzeichnis wird vom Handler dann als Wurzelverzeichnis dieses *ResourceRepository* freigegeben.
- **timeout** (default=10000): Millisekunden für den Http-Timeout beim Zugriff auf Daten.

- **methods** (default=GET): Eine komma-separierte Liste der Http-Methods *GET*, *PUT* und/oder *DELETE*, die auf diesem *ResourceRepository* erlaubt sein sollen. Dies kann natürlich auch eine Untermenge der Methoden sein, die vom eingesetzten Http-Server unterstützt werden!

➢ *JettyHttpResourceRepositoryHandler*

Dieser *ResourceRepositoryHandler* erweitert den *HttpResourceRepositoryHandler* um die Möglichkeit, den integrierten Jetty-Http-Server als Http-Server zur Verwaltung der Datenbasis einzusetzen. Er kann beispielsweise im Frontend verwendet werden, so daß dann auch das Backend über einen eigenen *HttpResourceRepositoryHandler* auf die Daten des gestarteten Jetty-Http-Servers zugreifen kann. In diesem Fall muß kein externer Http-Server vorhanden sein.

Der integrierte Http-Server, der hier verwendet wird, ist Jetty. Jetty unterstützt (ab Version 4) die PUT-Methode nach Spezifikation und kann daher auch zum Schreiben von Dateien und zum Anlegen von Verzeichnissen verwendet werden. Sofern notwendig/gewünscht kann der Schreibzugriff natürlich trotzdem eingeschränkt werden, indem man nur GET als erlaubte Methode definiert.

Neben den poolspezifischen Eigenschaften aus der Basis-Konfiguration wird dieser Handler durch die folgenden Optionen definiert:

Poolspezifische Eigenschaften

- **timeout**: wie beim *HttpResourceRepositoryHandler*.
- **methods**: wie beim *HttpResourceRepositoryHandler*.
- **host**: Der Hostname, unter dem auf Http-Anfragen gehört werden soll.
- **port**: Der Port, unter dem auf Http-Anfragen gehört werden soll.
- **standalone** (default=false): Bei *true* wird ein eigener Http-Server für dieses *ResourceRepository* gestartet, bei *false* teilt es sich mit anderen *ResourceRepository*-Diensten, die ebenfalls nicht standalone laufen, einen Http-Server.
- **minThreads** (default=1): Minimale Anzahl von Threads, die als Listener gestartet werden.
- **maxThreads** (default=5): Maximale Anzahl von Threads, die als Listener gestartet werden.
- **resourceBase**: Ein Pfad nach dem *file*-Protokoll, das auf die *Document Root* (Wurzelverzeichnis) für diesen Http-Server zeigt.
- **browsing** (default=false): Bei *true* ist Directory-Browsing erlaubt.

8.4.8 SessionManager-Service

Der *SessionManager*-Service wird vom Provider *SessionManagerServiceProvider* zur Verfügung gestellt. Er wird ausschließlich durch serviceglobale Eigenschaften konfiguriert. Die Eigenschaften werden alle nur in der Service-Konfiguration des Frontends bei Cluster Deployments und generell bei Single Container Deployments beachtet. Eine reine Backend-Konfiguration wird ohne spezielle Optionen vorgenommen.

Serviceglobale Eigenschaften

- **timeout** (default=unendlich): Nach Ablauf dieser Zeitspanne in Sekunden wird eine Session ungültig. Jede neue Anfrage für diese Session setzt den Zähler wieder auf 0 zurück.
- **checkinterval** (default=300): Zeitspanne in Sekunden, nach der die Session-Garbage-Collection jeweils abläuft, um alle ungültigen Sessions aus dem Gesamtsystem zu entfernen.
- **usecookies** (default=true): Sofern *true*, wird für jede neue Session versucht, das Session-Kennzeichen in einem Cookie abzulegen. Gelingt dies nicht, wird es in die Request-URI codiert. Bei *false* wird immer die Request-URI-Codierung verwendet.

8.4.9 ComponentSelector-Service

Der *ComponentSelector*-Service stellt den Zugriff auf SCF-Components zur Verfügung. Alle Components müssen dazu in der Deployment-Konfiguration definiert sein. Zur Anbindung des Connection-Service existiert nur der Provider
InstanceComponentSelectorServiceProvider
Dieser Provider verfügt über keine Konfigurationseigenschaften.

8.4.10 SCFManagement-Service

Der *SCFManagement*-Service ist ein System-Dienst, der der Anwendungsschicht keine Funktionalität bietet. Er muß trotzdem korrekt definiert werden, um die Synchronisation der Container eines *zen Deployments* untereinander sicherzustellen. Als Service-Provider existiert nur der *SCFManagementServiceProvider*. Er ist ausschließlich durch serviceglobale Eigenschaften definiert. Für jeden Container muß der zutreffende *SCFManagementHandler* konfiguriert werden. Alle momentan für die *zen Platform* spezifizierten Handler befinden sich im Package *de.zeos.scf.service.mgt.fecentered*.

Serviceglobale Eigenschaften

- **handler**: Der *SCFManagementHandler* ist abhängig von der Deployment-Variante bzw. dem Laufzeit-Container:
    - **Single Container Deployment**: *de.zeos.scf.service.mgt.fecentered.FrontendSCFManagementHandler*
    - **Frontend eines Cluster Deployments**: *de.zeos.scf.service.mgt.fecentered.FrontendSCFManagementHandler*
    - **Backend eines ClusterDeployments***:* *de.zeos.scf.service.mgt.fecentered.BackendSCFManagementHandler*

8.5 Start der zen-Engine

Um die *zen Engine* erfolgreich starten zu können, müssen neben der Deployment- und der Service-Konfiguration auch die notwendigen Klassenbibliotheken verfügbar und die Umgebungsvariablen korrekt gesetzt sein.

8.5.1 Klassenbibliotheken

Die *zen Engine* selbst besteht aus drei Klassenbibliotheken, die die Kernfunktionalität der *zen Platform* abbilden. Daneben setzt sie auf weitere Bibliotheken auf, die wie die Kern-Bibliotheken alle im Klassenpfad des jeweiligen Containers verfügbar sein müssen. Für den Start des Systems sind außerdem die Bibliotheken der *Tomcat Servlet-Engine* erforderlich. Je nach Service-Konfiguration müssen, unter anderem für die Datenbank, eventuell weitere Bibliotheken der jeweiligen Hersteller eingebunden werden.

Bibliothek	***Quelle***	***Unterstützte Versionen***	***Beschreibung***
zen-Basisbibliotheken			
scf.jar	zeos informatics	alle	Middeware der *zen Platform*
zencore.jar	zeos informatics	alle	*zen Engine*
zenapi.jar	zeos informatics	alle	zen API für die Anwendungsprogrammierung
Button Render System			
brs.jar	zeos informatics	alle	Dynamische Erstellung von Buttons/Images
Zusätzliche Bibliotheken			
J2ee.jar	Sun	1.3	Interfaces der J2ee-Spezifikation
kodo-jdo-runtime.jar	Solarmetric	3.1	JDO-Implementierung
Jakarta-commons-lang-1.0.1.jar	Apache Jakarta	Kodo-Package	Erweiterungen von java.lang
Jdo-1.0.1.jar	Sun	Kodo-Package	JDO-Spezifikation
minerva.jar	Jboss	Auslieferung	Pool-Implementierung von jboss
org.mortbay.jetty.jar	Jetty project	Auslieferung	Jetty-Http-Server
commons-logging.jar	Apache Jakarta	Auslieferung	Commons Logging API
log4J.jar	Apache Jakarta	Auslieferung	Log4J-Package
tyrex-1.0.jar	Tyrex project	Auslieferung	JNDI- und JTA-Implementierung
castor-0.9.4.3-xml.jar	Tyrex project	Tyrex-Package	Teil des Tyrex-Package
ots-jts_1.0.jar	Tyrex project	Tyrex-Package	Teil des Tyrex-Package (Corba OTS)
mx4j-jmx.jar	MX4J project	Auslieferung	JMX-Implementierung
mx4j-tools.jar	MX4J project	MX4J-Package	JMX-Implementierung
activation.jar	Sun	Auslieferung	JavaBeans Activation Framework
mail.jar	Sun	Auslieferung	Java-Mail-API
xalan.jar	Apache	Auslieferung	XSLT-Prozessor-Implementierung

Bibliothek	*Quelle*	*Unterstützte Versionen*	*Beschreibung*
xml-apis.jar	Apache	Auslieferung	Sammlung aller XML-spezifischen APIs
xercesImpl.jar	Apache	Auslieferung	XML-Parser-Implementierung
fop.jar	Apache	Auslieferung	FOP-Prozesser-Implementierung
avalon-framework.jar	Apache	Auslieferung	Apache-Framework
Tomcat Servlet-Engine			
bootstrap.jar	Apache	Auslieferung	
catalina.jar	Apache	Auslieferung	
commons-beansutils.jar	Apache	Auslieferung	
commons-collections.jar	Apache	Auslieferung	
commons-digester.jar	Apache	Auslieferung	
jasper-compiler.jar	Apache	Auslieferung	
jasper-runtime.jar	Apache	Auslieferung	
naming-common.jar	Apache	Auslieferung	
naming-resources.jar	Apache	Auslieferung	
servlet.jar	Apache	Auslieferung	
servlets-common.jar	Apache	Auslieferung	
servlets-default.jar	Apache	Auslieferung	
servlets-invoker.jar	Apache	Auslieferung	
tomcat-coyote.jar	Apache	Auslieferung	
tomcat-http11.jar	Apache	Auslieferung	
tomcat-util.jar	Apache	Auslieferung	

Datenbank-Bibliotheken

Neben den Standard-Bibliotheken müssen die Java-Treiber der verwendeten Datenbank zusätzlich im Klassenpfad der *zen Engine* liegen.

8.5.2 Startparameter

Für den erfolgreichen Start der *zen Engine* sind einige Startparameter unbedingt erforderlich, andere sind optional und abhängig vom Container. Die Konfiguration der Parameter muß zum Teil als Umgebungsparameter erfolgen, andere Parameter werden als Properties in der Konfigurationsdatei

zen.properties

definiert, die im Klassenpfad der jeweiligen JVM abgelegt sein muß.

➢ *Logging*

LogFactory

Die LogFactory definiert den Einsprungpunkt der *zen Platform* in die *Apache Jakarta Commons Logging* Bibliotheken. Dieser Parameter muß unbedingt als Umgebungsparameter beim Start der JVM gesetzt sein. Die Property muß fest als

org.apache.commons.logging.LogFactory=de.zeos.scf.service.log.SCFLogFactory

definiert werden. Sofern die Klasse *de.zeos.zen.tomcat.TomcatStart* aus der *zen* Platform zum Starten der *zen Engine* im Frontend-Container verwendet wird, erledigt sie dies automatisch. Bei Backend-Containern bzw. Applikationsservern muß die Variable beim Start der JVM mit

-D*org.apache.commons.logging.LogFactory=de.zeos.scf.service.log.SCFLogFactory*

als Kommandozeilenparameter gesetzt werden.

Logging-Subsystem (optional)

Das Logging-Subsystem, über das die Logausgabe erfolgen soll, wird in *zen.properties* als

de.zeos.scf.service.log.system=jdk

bzw.

de.zeos.scf.service.log.system=log4j

konfiguriert. Ist der Wert nicht definiert, wird automatisch die Java Logging API verwendet. Dies setzt jedoch Java 1.4 voraus.

➢ *Container-Erkennung*

Während dem Hochfahren einer *zen Engine* muß sich der jeweilige Container autonom anhand der zentralen Deployment-Konfiguration identifizieren. Durch die Deployment-Konfiguration ist auch vorgegeben, ob ein Container als Frontend oder Backend fungieren soll und welchem Island und Cluster er zugeordnet ist.

Für die Identifikation muß im jeweiligen Container die Property
de.zeos.scf.deploy.container.name
mit dem Namen des Containers aus der Deployment-Konfiguration gesetzt sein. Die Property kann dazu entweder als Property des *InitialContext* definiert werden (üblicherweise in der Datei *jndi.properties*) oder sie muß als Systemvariable beim Start der jeweiligen JVM gesetzt werden.

➢ *DeploymentConfigurationFactory*

Das Setzen der *DeploymentConfigurationFactory* wird über den Parameter
de.zeos.scf.deploy.factory.initial
in *zen.properties* des jeweiligen Laufzeit-Containers vorgenommen. Erst durch die angegebene Factory wird spezifiziert, aus welcher Quelle die Deployment-Konfiguration gelesen wird.

Für das Frontend mit Dateizugriff wird diese Property dann als
de.zeos.scf.deploy.factory.initial=
de.zeos.scf.deploy.config.file.FileBasedDeploymentConfigurationFactory
gesetzt.

➢ *ServiceConnectorFactory*

Das Setzen der *ServiceConnectorFactory* wird über den Parameter
de.zeos.scf.service.factory.initial
in *zen.properties* des jeweiligen Laufzeit-Containers vorgenommen. Erst durch die angegebene Factory wird spezifiziert, aus welcher Quelle die Service-Konfiguration gelesen wird.

Für das Frontend mit Dateizugriff wird diese Property dann als
de.zeos.scf.service.factory.initial=
de.zeos.scf.service.config.file.FileBasedServiceConnectorFactory
gesetzt.

➢ *Contexthandler*

Der Contexthandler wird in *zen.properties* als
de.zeos.scf.contexthandler
definiert. Für Single Container Deployments und für das Frontend bei Cluster Deployments muß immer
de.zeos.scf.contexthandler=de.zeos.scf.container.TyrexContextHandler
gesetzt sein. Für reine Backends wird immer
de.zeos.scf.contexthandler=de.zeos.scf.container.DefaultContextHandler
definiert.

➢ *Administration*

Alle Administrationsparameter werden direkt in *zen.properties* angegeben.

de.zeos.init.jmxhttpport

Dieser Parameter wird nur im Frontend-Container notwendig. Er definiert den Port, über den die zentrale Administrationskonsole erreichbar ist. Im Backend ist keine Konfiguration notwendig.

Der Zugriff auf die zentrale Administrationskonsole über Port 8085 wird dann konfiguriert als
de.zeos.init.jmxhttpport=8085

➢ *Tomcat-Steuerung*

Sofern die Klasse *de.zeos.zen.tomcat.TomcatStart* aus der *zen Platform* zum Starten der *zen Engine* verwendet wird, können verschiedene optionale Parameter zur Tomcat-Konfiguration direkt in *zen.properties* angegeben werden:

de.zeos.init.catalina.host (Default: localhost)

Der Hostname, unter dem die Tomcat auf eingehene Aufrufe...

de.zeos.init.catalina.port (Default: 8080)

...unter dem angegebenen Port hört.

de.zeos.init.catalina.home (Default: Arbeitsverzeichnis)

Das Home-Verzeichnis der Tomcat-Installation. Sofern keines angegeben wird, wird im Arbeitsverzeichnis ein Unterverzeichnis *tomcat* mit den notwendigen Dateien angelegt.

de.zeos.init.catalina.ctx (Default: /zen/*)

Der Kontextpfad, unter dem die *zen Engine* in der Tomcat ausgeführt werden soll.

➢ *Repository*

Zur Konfiguration des Repository können folgende optionale Parameter in *zen.properties* gesetzt werden.

de.zeos.zen.ext.frontend.generic.repository

Wenn das *GenericServlet* verwendet wird, kann unter dieser Property der Name des Standard-Repository angegeben werden, in dem die jeweilige Anwendung gesucht wird.

➢ *WebServices*

Das Wurzel-Verzeichnis, unter dem bei Web-Service-Anfragen alle Referenzen der entsprechenden Schema-Datei aufgelöst werden, wird über die Property
de.zeos.init.xsd.fomhook.systemid
in *zen.properties* definiert.

Anhang

A Konfiguration Deployment

A.1 DTD scfdeploy.xml

Die Datei *scfdeploy.xml* für die Deployment-Konfiguration muß folgender DTD genügen:

```
<!DOCTYPE scfdeploy [

<!ELEMENT scfdeploy (containers,clusters,islands,components)>

<!ELEMENT containers (container)*>
<!ELEMENT clusters   (cluster)*>
<!ELEMENT islands    (island)*>
<!ELEMENT components (component)*>

<!ELEMENT container (property)*>
<!ATTLIST container name CDATA #REQUIRED>
<!ATTLIST container mnemonic CDATA #IMPLIED>
<!ATTLIST container sticky (true|false) "false">
<!ATTLIST container remoteAccess (true|false) "false">
<!ATTLIST container acceptingNewSessions (true|false) "true">

<!ELEMENT property (#PCDATA)>
<!ATTLIST property name CDATA #REQUIRED>

<!ELEMENT cluster (filter?,islandrefs)>
<!ATTLIST cluster name CDATA #REQUIRED>
<!ATTLIST cluster mnemonic CDATA #IMPLIED>


<!ELEMENT islandrefs (islandref)*>

<!ELEMENT islandref EMPTY>
<!ATTLIST islandref name CDATA #REQUIRED>

<!ELEMENT island (filter?,containerrefs)>
<!ATTLIST island name CDATA #REQUIRED>
<!ATTLIST island mnemonic CDATA #IMPLIED>
<!ATTLIST island failover (true|false) #REQUIRED>
<!ATTLIST island alwaysAcceptingNewSessions (true|false) "false">

<!ELEMENT containerrefs (containerref)*>

<!ELEMENT containerref EMPTY>
<!ATTLIST containerref name CDATA #REQUIRED>

<!ELEMENT filter EMPTY>
<!ATTLIST filter class CDATA #REQUIRED>

<!ELEMENT component (runtimeclass,targetcluster,monitoring?,journaling?)>
<!ATTLIST component name CDATA #REQUIRED>

<!ELEMENT runtimeclass (#PCDATA)>

<!ELEMENT targetcluster (#PCDATA)>
<!ATTLIST targetcluster pooled (true|false) "false">
<!ATTLIST targetcluster min CDATA #IMPLIED>
<!ATTLIST targetcluster max CDATA #IMPLIED>
<!ATTLIST targetcluster recycle (true|false) "false">

<!ELEMENT monitoring EMPTY>
<!ATTLIST monitoring alarmTimeLimit CDATA #IMPLIED>
<!ATTLIST monitoring alarmThresholdIncidents CDATA #IMPLIED>
<!ATTLIST monitoring alarmThresholdWindow CDATA #IMPLIED>

<!ELEMENT journaling EMPTY>
<!ATTLIST journaling slots CDATA #IMPLIED>
<!ATTLIST journaling slotTime CDATA #IMPLIED>

]>
```

A.1.1 Beispielkonfiguration: Single Container Deployment

Dies ist ein Beispiel für die minimale Deployment-Konfiguration *scfdeploy.xml* eines Single Container Deployments ohne DTD. Die zentrale JMX-Management-Console für das Frontend ist auf dem Port 8085 definiert:

```
<scfdeploy>

    <containers>
        <container name="single" mnemonic="fe" remoteAccess="false">
            <property name="java.naming.factory.initial">
                    tyrex.naming.MemoryContextFactory</property>
            <property name="java.naming.factory.url.pkgs">
                    tyrex.naming</property>
            <property name="de.zeos.scf.deploy.config.jmx.url">
                    http://localhost:8085</property>
        </container>
    </containers>

    <islands>
        <island name="single_island" mnemonic="ife" failover="false">
            <containerrefs>
                <containerref name="single"/>
            </containerrefs>
        </island>
    </islands>

    <clusters>
        <cluster name="single_cluster" mnemonic="cfe">
            <islandrefs>
                <islandref name="single_island"/>
            </islandrefs>
        </cluster>
    </clusters>

    <components>
        <component name="Kernel">
            <runtimeclass>de.zeos.zen.core.comp.Kernel</runtimeclass>
            <targetcluster>single_cluster</targetcluster>
        </component>
        <component name="XSLProcessor">
            <runtimeclass>de.zeos.zen.ext.comp.XSLProcessor</runtimeclass>
            <targetcluster>single_cluster</targetcluster>
        </component>
    </components>

</scfdeploy>
```

A.1.2 Beispielkonfiguration für Cluster Deployment

➢ *Cluster Deployment mit einem Backend (jboss)*

Dies ist ein Beispiel für eine bis auf den FOP-Prozessor minimale Deployment-Konfiguration *scfdeploy.xml* eines Cluster Deployments ohne DTD. Neben dem Frontend ist ein JBoss-Applikationsserver (v 2.4.x) auf einem Server mit dem Namen *appserver* konfiguriert. JBoss stellt über Port 1099 den externen JNDI-Zugriff zur Verfügung.

Die zentrale JMX-Management-Console für das Frontend ist auf dem Port 8085 definiert. Auf die JMX-Console des Applikationsservers kann man über Port 8082 auf *appserver* zugreifen.

Der FOP-Prozessor ist auf dem Frontend in einem Pool verfügbar und wird für die PDF-Ausgabe genutzt.

```
<scfdeploy>

    <containers>
        <container name="frontend" mnemonic="fe" remoteAccess="false">
            <property name="java.naming.factory.initial">
                    tyrex.naming.MemoryContextFactory</property>
            <property name="java.naming.factory.url.pkgs">
                    tyrex.naming</property>
            <property name="de.zeos.scf.deploy.config.jmx.url">
                    http://localhost:8085</property>
        </container>
        <container name="jboss_backend" mnemonic="be" remoteAccess="true">
            <property name="java.naming.provider.url">
                    appserver:1099</property>
            <property name="java.naming.factory.initial">
                    org.jnp.interfaces.NamingContextFactory</property>
            <property name="java.naming.factory.url.pkgs">
                    org.jboss.naming:org.jnp.interfaces</property>
            <property name="de.zeos.scf.deploy.config.jmx.url">
                    http://appserver:8082</property>
        </container>
    </containers>

    <islands>
        <island name="island_frontend" mnemonic="ife" failover="false">
            <containerrefs>
                <containerref name="frontend"/>
            </containerrefs>
        </island>
        <island name="island_backend" mnemonic="ibe" failover="false">
            <containerrefs>
                <containerref name="jboss_backend"/>
            </containerrefs>
        </island>
    </islands>

    <clusters>
        <cluster name="cluster_frontend" mnemonic="cfe">
            <islandrefs>
                <islandref name="island_frontend"/>
            </islandrefs>
        </cluster>
        <cluster name="cluster_backend" mnemonic="cbe">
            <islandrefs>
                <islandref name="island_backend"/>
```

```
            </islandrefs>
        </cluster>
    </clusters>

    <components>
        <component name="Kernel">
            <runtimeclass>de.zeos.zen.core.comp.Kernel</runtimeclass>
            <targetcluster>cluster_backend</targetcluster>
        </component>
        <component name="XSLProcessor">
            <runtimeclass>de.zeos.zen.ext.comp.XSLProcessor</runtimeclass>
            <targetcluster>cluster_frontend</targetcluster>
        </component>
        <component name="FOPProcessor">
            <runtimeclass>de.zeos.zen.ext.comp.XSLProcessor</runtimeclass>
            <targetcluster pooled="true" min="3" max="7">cluster_frontend</targetcluster>
        </component>
    </components>

</scfdeploy>
```

➢ *Cluster Deployment mit zwei Backends (JBoss)*

Dies ist ein weiteres Beispiel für eine Deployment-Konfiguration *scfdeploy.xml* eines Clusters ohne DTD. Neben dem Frontend sind zwei JBoss-Applikationsserver (v 2.4.x) auf zwei Servern mit den Namen *appserver1* und *appserver2* konfiguriert. JBoss stellt auf jedem Server jeweils über Port 1099 den externen JNDI-Zugriff zur Verfügung.

Im Backendcluster ist für das einzige Island die Session-Replikation beim Start aktiviert.

Die zentrale JMX-Management-Console für das Frontend ist auf dem Port 8085 definiert. Auf die JMX-Console der Applikationsservers kann man jeweils über Port 8082 zugreifen.

Der XSL-Prozessor wird sowohl innerhalb des Frontends für die Ausgabe-Transformation genutzt, als auch von der Geschäftslogik des Backends für interne Transformationen genutzt. Daher liegt er nicht in einem expliziten Cluster.

```
<scfdeploy>

    <containers>
        <container name="frontend" mnemonic="fe" remoteAccess="false">
            <property name="java.naming.factory.initial">
                    tyrex.naming.MemoryContextFactory</property>
            <property name="java.naming.factory.url.pkgs">
                    tyrex.naming</property>
            <property name="de.zeos.scf.deploy.config.jmx.url">
                    http://localhost:8085</property>
        </container>
        <container name="jboss_backend_1" mnemonic="be1" remoteAccess="true">
            <property name="java.naming.provider.url">
                    appserver1:1099</property>
            <property name="java.naming.factory.initial">
                    org.jnp.interfaces.NamingContextFactory</property>
            <property name="java.naming.factory.url.pkgs">
                    org.jboss.naming:org.jnp.interfaces</property>
            <property name="de.zeos.scf.deploy.config.jmx.url">
                    http://appserver1:8082</property>
        </container>
        <container name="jboss_backend_2" mnemonic="be2" remoteAccess="true">
            <property name="java.naming.provider.url">
                    appserver2:1099</property>
            <property name="java.naming.factory.initial">
                    org.jnp.interfaces.NamingContextFactory</property>
            <property name="java.naming.factory.url.pkgs">
                    org.jboss.naming:org.jnp.interfaces</property>
            <property name="de.zeos.scf.deploy.config.jmx.url">
                    http://appserver2:8082</property>
        </container>
    </containers>

    <islands>
        <island name="island_frontend" mnemonic="ife" failover="false">
            <containerrefs>
                <containerref name="frontend"/>
            </containerrefs>
        </island>
        <island name="island_backend" mnemonic="ibe" failover="true">
            <containerrefs>
                <containerref name="jboss_backend_1"/>
                <containerref name="jboss_backend_2"/>
            </containerrefs>
        </island>
    </islands>

    <clusters>
        <cluster name="cluster_frontend" mnemonic="cfe">
            <islandrefs>
                <islandref name="island_frontend"/>
            </islandrefs>
        </cluster>
        <cluster name="cluster_backend" mnemonic="cbe">
            <islandrefs>
                <islandref name="island_backend"/>
            </islandrefs>
        </cluster>
    </clusters>

    <components>
        <component name="Kernel">
            <runtimeclass>de.zeos.zen.core.comp.Kernel</runtimeclass>
            <targetcluster>cluster_backend</targetcluster>
        </component>
        <component name="XSLProcessor">
            <runtimeclass>de.zeos.zen.ext.comp.XSLProcessor</runtimeclass>
```

```
            <targetcluster/>
        </component>
    </components>

</scfdeploy>
```

A.1.3 Beispielkonfiguration für Monitoring & Journaling

Die Beispielkonfiguration zeigt nur einen notwendigen Ausschnitt aus der Deployment-Konfiguration *scfdeploy.xml*. Die Component *Kernel* wird hier einmal überwacht, so daß ein Alarm über den Monitoring-Logger ausgelöst wird, wenn 5 Aufruf-Zyklen innerhalb von 10 Sekunden länger als 2,5 Sekunden dauern. Gleichzeitig werden alle Aufrufe in 10 Slots protokolliert, wobei ein Slot genau 1 Minute lang dauert.

```
<scfdeploy>

    ...

    <components>
        <component name="Kernel">
            <runtimeclass>de.zeos.zen.core.comp.Kernel</runtimeclass>
            <targetcluster>cluster_backend</targetcluster>
            <monitoring alarmTimeLimit="2500"
                        alarmThresholdIncidents="5"
                        alarmThresholdWindow="10000"/>
            <journaling slots="10" slotTime="PT1M"/>
        </component>
    </components>

</scfdeploy>
```

B Konfiguration Service

Alle Klassenpfade, die mit *de.zeos.scf.service* beginnen, werden der Übersichtlichkeit halber mit *dzss* abgekürzt.

B.1 DTD scfservice.xml

```
<!DOCTYPE scfservice [

<!  root jndi-path where the provider should store references to, e.g. 'comp/env/jdbc' -->
<!ENTITY BASICPROPS_JNDIACCESS "jndiaccess">

<!  comma separated string of classes that must be loaded before the
    service is initialized, e.g. jdbc-drivers -->
<!ENTITY BASICPROPS_PRELOAD "preload">

<!-- definition of the different log levels of java logging api. -->
<!ENTITY LEVEL.OFF         "OFF">
<!ENTITY LEVEL.SEVERE      "SEVERE">
<!ENTITY LEVEL.WARNING     "WARNING">
<!ENTITY LEVEL.INFO        "INFO">
<!ENTITY LEVEL.CONFIG      "CONFIG">
<!ENTITY LEVEL.FINE        "FINE">
<!ENTITY LEVEL.FINER       "FINER">
<!ENTITY LEVEL.FINEST      "FINEST">
<!ENTITY LEVEL.ALL         "ALL">

<!ELEMENT scfservice (  Logging,Mail,SessionManager,Transaction,Connection,
                        SCFManagement,JDO,Messaging,ComponentSelector,ResourceRepository)>

<!ELEMENT Logging (properties?, pools?)>
<!ATTLIST Logging provider CDATA #REQUIRED>

<!ELEMENT Mail (properties?, pools?)>
<!ATTLIST Mail provider CDATA #REQUIRED>

<!ELEMENT SessionManager (properties?, pools?)>
<!ATTLIST SessionManager provider CDATA #REQUIRED>

<!ELEMENT Transaction (properties?, pools?)>
<!ATTLIST Transaction provider CDATA #REQUIRED>

<!ELEMENT Connection (properties?, pools?)>
<!ATTLIST Connection provider CDATA #REQUIRED>

<!ELEMENT SCFManagement (properties?, pools?)>
<!ATTLIST SCFManagement provider CDATA #REQUIRED>

<!ELEMENT JDO (properties?, pools?)>
<!ATTLIST JDO provider CDATA #REQUIRED>

<!ELEMENT Messaging (properties?, pools?)>
<!ATTLIST Messaging provider CDATA #REQUIRED>

<!ELEMENT ComponentSelector (properties?, pools?)>
<!ATTLIST ComponentSelector provider CDATA #REQUIRED>

<!ELEMENT ResourceRepository (properties?, pools?)>
<!ATTLIST ResourceRepository provider CDATA #REQUIRED>

<!ELEMENT properties (property)*>
<!ELEMENT property (#PCDATA)>
<!ATTLIST property name CDATA #REQUIRED>

<!ELEMENT pools (pool)*>
<!ELEMENT pool (property)*>
<!ATTLIST pool name CDATA #REQUIRED>

]>
```

B.1.1 Beispielkonfiguration Logging-Service

Das Beispiel stellt nur den Bereich des Logging-Service aus der Service-Konfiguration dar.

```
<!-- Für den Logging-Service gibt es nur diesen Provider-->
<Logging provider="dzss.provider.LoggingServiceProvider">

<!--serviceglobale Eigenschaften-->
<properties>
    <!--.level definiert den globalen Log-Level-->
    <property name=".level">&LEVEL.INFO;</property>
    <!--der globale Handler ist ein ExtConsoleHandler, außerdem existiert ein Enhanced
        FileHandler um die Ausgabe zu speichern und ein zusätzlicher MailHandler für
        schwerwiegende Fehler-->
    <property name=".handlers">
                    dzss.log.jdk.ExtConsoleHandler,
                    dzss.log.jdk.EnhancedFileHandler,
                    dzss.log.jdk.JdkMessagingMailHandler</property>
    <!--Der globale ExtConsoleHandler loggt alles, was mindestens den Level fine hat-->
    <property name="dzss.log.jdk.ExtConsoleHandler.level">&LEVEL.FINE;</property>
    <!--Der ExtConsoleHandler soll mit seinem 1-Argument-Konstruktor instanziiert werden,
        dem dabei eine Instanz eines SimpleLineFormatters übergeben wird-->
    <property name=
        "dzss.log.jdk.ExtConsoleHandler.construct.1.java.util.logging.Formatter">
            class:dzss.log.jdk.SimpleLineFormatter</property>
    <!--Der EnhancedFileHandler loggt alles-->
    <property name="dzss.log.jdk.EnhancedFileHandler.level">&LEVEL.ALL;</property>
    <!--in die angegebene Datei. Rollover-Indizes werden anstelle von %g eingesetzt-->
    <property name="dzss.log.jdk.EnhancedFileHandler.construct.1.java.lang.String">
        /zeos/log/completeoutput%g.log</property>
    <!--Die max. Größe einer Logdatei sind 204800 Bytes-->
    <property name="dzss.log.jdk.EnhancedFileHandler.construct.2.int">2048000</property>
    <!--Es sollen maximal 10 Rollover-Dateien genutzt werden-->
    <property name="dzss.log.jdk.EnhancedFileHandler.construct.3.int">10</property>
    <!--Falls schon eine Log-Datei existiert, sollen die Logeinträge angehängt werden-->
    <property name="dzss .log.jdk.EnhancedFileHandler.construct.4.boolean">true</property>
    <!-- Der EnhancedFileHandler soll den gleichen Formatter wie oben anwenden-->
    <property
        name="dzss.log.jdk.EnhancedFileHandler.construct.5.java.util.logging.Formatter">
            class:dzss.log.jdk.SimpleLineFormatter</property>
    <!--Der MailHandler versendet nur Logeinträge, die als schwerwiegend
        gekennzeichnet sind-->
    <property name="dzss.log.jdk.JdkMessagingMailHandler.level">&LEVEL.SEVERE;</property>
    <!--Der Empfänger des Mail-Logs ist hier angegeben-->
    <property name="dzss .log.jdk.JdkMessagingMailHandler.construct.1.java.util.ArrayList">
        developer@localhost</property>
    <!--Dem Namen des Container, in dem der Logeintrag erstellt wird, soll der angegebene
        String vorangestellt werden-->
    <property name="dzss.log.jdk.JdkMessagingMailHandler.construct.2.java.lang.String">
        localhost-tomcat</property>

</properties>

<pools>

    <!--Der SCF-Logger nutzt nur die globalen Handler mit deren Log Level-->
    <pool name=“scf“>
        <property name=".useglobalhandlers">true</property>
    </pool>

    <!--Der myown-Logger schreibt alles ab Level WARNING nur in seine eigene Log-Datei
        über den Java-Standard-FileHandler. Dadurch gehen alle Logeinträge kleiner als
        WARNING verloren!-->
    <pool name=“myown“>
        <property name=
                “.handlers“>java.util.logging.FileHandler.level“>&LEVEL.WARNING;</property>
        <property name=".useglobalhandlers">false</property>
        <property name=
                "java.util.logging.FileHandler.construct.1.java.lang.String">
                    /zeos/log/myown%g.log</property>
        <property name="java.util.logging.FileHandler.construct.2.int">1024000</property>
        <property name="java.util.logging.FileHandler.construct.3.int">10</property>
        <property name="java.util.logging.FileHandler.construct.4.boolean">true</property>
    </pool>

</pools>

</Logging>
```

B.1.2 Beispielkonfiguration Mail-Service

Das Beispiel stellt nur den Bereich des Mail-Service aus der Service-Konfiguration dar.

```
<Mail provider="dzss.provider.MailServiceJNDIProvider">

    <!--serviceglobale Eigenschaften-->
    <properties>
        <!  Die Session wird damit als comp/env/mail/MailSession im JNDI plaziert-->
        <property name="jndisessionname">MailSession</property>
    </properties>

</Mail>
```

B.1.3 Beispielkonfiguration Transaction-Service

Das Beispiel stellt nur den Bereich des Transaction-Service aus der Service-Konfiguration dar.

```
<Transaction provider="dzss.provider.TransactionServiceJNDIProvider">

    <properties>
        <property name="tmjndiinstancename">TransactionManager</property>
        <property name="utxjndiinstancename">UserTransaction</property>
    </properties>
```

```
</Transaction>
```

B.1.4 Beispielkonfiguration Connection-Service

Das Beispiel stellt nur den Bereich des Connection-Service aus der Service-Konfiguration dar.

```
<Connection provider="dzss.provider.ConnectionServiceJNDIProvider">

<properties>
    <property name="&BASICPROPS_PRELOAD;">oracle.jdbc.driver.OracleDriver</property>
</properties>

<pools>
    <!--Ein Oracle-Pool mit echten Connections für die Anbindung eines Repositories-->
    <pool name="test.repository">
        <property name="jndiinstancename">test.repository</property>
        <property name="url">jdbc:oracle:thin:@dbserver:1521:zen</property>
        <property name="user">test</property>
        <property name="password">test</property>
        <property name="min">1</property>
        <property name="max">10</property>
        <property name="blocking">true</property>
    </pool>
    <!--Ein XA-Oracle-Pool mit echten Connections, der auch dem
        Transaktionsmanager bekannt gemacht wird-->
    <pool name="test.persist">
        <property name="jndiinstancename">test.persist</property>
        <property name="url">jdbc:oracle:thin:@dbserver:1521:zen</property>
        <property name="user">testpersist</property>
        <property name="password">testpersist</property>
        <property name="min">1</property>
        <property name="max">10</property>
        <property name="transactional">true</property>
        <property name="xadriver">de.zeos.scf.core.db.XADataSourceOracleImpl</property>
    </pool>
    <!--Ein virtueller Pool, der die Connections von test.persist mitnutzt-->
    <pool name="example">
        <property name="jndiinstancename">test.persist</property>
    </pool>

</Connection>
```

B.1.5 Beispielkonfiguration JDO-Service

Das Beispiel stellt nur den Bereich des JDO-Service aus der Service-Konfiguration dar.

```
<JDO provider="dzss.provider.JDOServiceJNDIProvider">

<pools>
    <!--Das Repository test.repository setzt auf den gleichnamigen Connection-Pool
        test.repository auf-->
    <pool name="test.repository">
        <property name="connectionservice">test.repository</property>
    </pool>
</pools>

</JDO>
```

B.1.6 Beispielkonfiguration Messaging-Service

Das Beispiel stellt nur den Bereich des Messaging-Service aus der Service-Konfiguration dar.

```
<Messaging provider="dzss.provider.MessagingServiceProvider">

<pools>
    <!--Das ist die lokale Konfiguration des MailMessageLoggers, der die LogMails aus
        einer simulierten Queue holt und an die Empfänger weiterleitet-->
    <pool name="MailMessageLogger">
        <property name="handler">dzss.msg.MessengerLocalImpl</property>
        <property name="timeout">75</property>
        <property name="queueName">queue/MailMessageLogger</property>
        <property name="systemName">dzss.log.MailMessageLogger</property>
    </pool>
    <!--Die lokale Konfiguration eines Prozesses, der Beispielsweise als
        Hintergrundprozeß parallele Arbeiten durchführen könnte-->
    <pool name="ParallelProcessing">
        <property name="handler">de.zeos.scf.service.msg.MessengerLocalImpl</property>
        <property name="timeout">75</property>
        <property name="queueName">queue/ParallelProcessing</property>
        <property name="systemName">de.my.bean.ParallelProcessingMessageBean</property>
    </pool>
</pools>

</Messaging>
```

B.1.7 Beispielkonfiguration ResourceRepository-Service

Das Beispiel stellt nur den Bereich des *ResourceRepository*-Service aus der Service-Konfiguration dar.

```
<ResourceRepository provider="dzss.provider.ResourceRepositoryServiceProvider">

<pools>
    <!--Ein Pool mit Datenzugriff auf das angegebene Verzeichnis-->
    <pool name="xsl">
        <property name="handler">dzss.res.FileResourceRepositoryHandler</property>
        <property name="url">file:///zeos/develop/xslpool/default/</property>
    </pool>
    <!--Ein Pool, der einen Jetty-Http-Server startet und über Http auf diesen lesend und
        schreibend zugreifen kann. Das Wurzelverzeichnis ist die resourceBase.-->
    <pool name="my-resources">
```

```
        <property name="handler">dzss.res.JettyHttpResourceRepositoryHandler</property>
        <property name="standalone">true</property>
        <property name="host">localhost</property>
        <property name="port">7878</property>
        <property name="resourceBase">file:///zeos/develop/xslpool/special/</property>
        <property name="contextPath">/</property>
        <property name="methods">GET, PUT</property>
    </pool>
</pools>

</ResourceRepository>
```

B.1.8 Beispielkonfiguration SessionManager-Service

Das Beispiel stellt nur den Bereich des *SessionManager*-Service aus der Service-Konfiguration dar.

```
<SessionManager provider="dzss.provider.SessionManagerServiceProvider">

<properties>
    <property name="timeout">750</property>
    <property name="checkinterval">750</property>
    <property name="usecookies">true</property>
</properties>

</SessionManager>
```

B.1.9 Beispielkonfiguration ComponentSelector-Service

Das Beispiel stellt nur den Bereich des *ComponentSelector*-Service aus der Service-Konfiguration dar.

```
<ComponentSelector provider="dzss.provider.InstanceComponentSelectorServiceProvider">
</ComponentSelector>
```

B.1.10 Beispielkonfiguration SCFManagement-Service

Das Beispiel stellt nur den Bereich des *SCFManagement*-Service aus der Service-Konfiguration dar.

```
<SCFManagement provider="dzss.provider.SCFManagementServiceProvider">

<properties>
    <property name="handler">dzss.mgt.fecentered.FrontendSCFManagementHandler</property>
</properties>

</SCFManagement>
```