Ein Framework zur modularisierten und pattern-basierten Entwicklung von zuverlässigen, personalisierten, web-basierten Applikationen

Dissertation

zur Erlangung des Grades eines

D o k t o r s d e r N a t u r w i s s e n s c h a f t e n

der Universität Dortmund
am Fachbereich Informatik
von

Claudia Gsottberger

Dortmund

2004

Tag der mündlichen Prüfung: 8.3.2005

Dekan: Prof. Dr. Bernhard Steffen

Gutachter: Prof. Dr. Bernhard Steffen
Prof. Dr. Ernst-Erich Doberkat

Danksagung

Viele Personen haben auf die eine oder andere Weise Anteil am Gelingen dieser Arbeit.

Besonders bedanken möchte ich mich bei Prof.Dr. Bernhard Steffen, der mich seit meinem Studium betreut und begleitet und mir durch die Übergabe der Projektleitung für das WIS-Teilprojekt 'Koordination des Online-Angebots des Fachbereichs Informatik' die Promotion in diesem Themenbereich erst ermöglicht hat.

Bei Dr. Volker Braun möchte ich mich für die zahlreichen und zeitintensiven Diskussionen bedanken, durch die er mir ganz neue Einsichten vermitteln konnte. Sein Feedback ist an vielen Stellen in die Arbeit eingeflossen und hat sie dadurch bereichert.

Auch Dr. Oliver Niese, Dr. Stefan Dißmann, PD Dr. Markus Müller-Olm haben mit wertvollen Anregungen meinen Blick für die Berücksichtigung neuer Aspekte geschärft. Bei ihnen möchte ich mich für die zahlreichen hilfreichen Gespräche und die geduldige Beantwortung meiner Fragen bedanken.

Markus Bajohr, Simon Muras, Andreas Dimek, Peter Lammich haben als Entwickler, Andreas Holzmann und Martin Karusseit als Berater entscheidend an der Realisierung des in dieser Arbeit beschriebenen Frameworks und der Beispiel-Applikation TeMPlUs mitgewirkt. Ihnen möchte ich für ihren Einsatz und Teamgeist sowie die motivierende und kreative Arbeitsatmosphäre danken.

Bedanken möchte ich mich auch bei Prof.Dr.E.-E. Doberkat, der sich bereit erklärt hat, Zweitgutachter für meine Arbeit zu sein.

Ganz besonderer Dank gebührt meinen Eltern, die mir während schwieriger Zeiten immer die bestmögliche Unterstützung haben zukommen lassen und mir den Blick für das Wesentliche geöffnet haben.

Inhaltsverzeichnis

Abbildungsverzeichnis

Teil I

Motivation und Hintergrund

Kapitel 1

Ausgangssituation

Mit dem *Sonderprogramm zur Weiterentwicklung des Informatikstudiums an den deutschen Hochschulen*[1] *(WIS)* werden Maßnahmen zur Steigerung von Effizienz, Niveau und Betreuung der Informatikausbildung entwickelt. Gegenstand der Förderung sind insbesondere Maßnahmen zur Schaffung zusätzlicher Ausbildungskapazitäten, zur Verkürzung der Studienzeiten und zur Entwicklung/Erprobung neuer Studiengänge mit den Abschlüssen Bachelor und Master sowie von Studienangeboten der Weiterbildung an Hochschulen.

Im Rahmen dieses Sofortprogramms wurde am Fachbereich Informatik der Universität Dortmund ein lehrstuhlübergreifendes Projekt (WIS-Projekt) zur Steigerung der Erfolgsquote im Studium initiiert. Die grundlegende Zielsetzung besteht dabei – wie in [98] ausführlich beschrieben – in der Unterstützung und Ergänzung der klassischen Lehre durch eine Rosette von Maßnahmen aus verschiedenen Bereichen, wie in Abb. 1.1 skizziert. Dadurch soll eine Verbesserung der Lehresituation erreicht und somit der Erfolg der Studierenden im Informatikstudium möglich bzw. beschleunigt werden.

Um die Lehresituation an einer Universität zu verbessern, gibt es grundsätzlich verschiedene Möglichkeiten. An der Universität Dortmund existieren mehrere WIS-Teilprojekte, die auf unterschiedliche Art und Weise versuchen, dieses Ziel zu erreichen.

Das WIS-Teilprojekt Nr. 6[2] beschäftigt sich mit der Entwicklung von Konzepten für die 'Koordination des Online-Angebots des Fachbereichs Informatik' und lässt sich in die Maßnahmen-Rosette des gesamten WIS-Projektes an der Universität Dortmund wie folgt einordnen (siehe Abb. 1.1). Dabei unterstützt das WIS-Teilprojekt Nr. 6

[1] Dies wurde in der Bund-Länder Vereinbarung vom 19.6.2000 festgelegt.

[2] Das WIS-Teilprojekt Nr. 6 wurde am Lehrstuhl Informatik 5 der Universität Dortmund unter der Leitung von Prof. Dr. B. Steffen und Dipl.-Inform. C. Gsottberger durchgeführt.

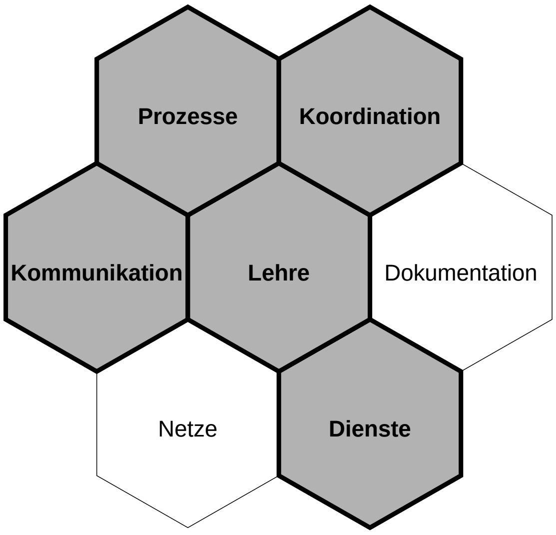


Abbildung 1.1: Einordnung des WIS-Teilprojekts Nr. 6 in die Maßnahmen-Rosette

die klassische Lehre – in Abb. 1.1 zentral dargestellt – mit Maßnahmen aus den um sie angeordneten, hervorgehobenen Bereichen. Wir unterteilen diese Maßnahmen in konzeptionelle Aspekte und deren technische Umsetzung.

Folgende konzeptionellen Aspekte sind im WIS-Teilprojekt Nr. 6 von zentraler Bedeutung:

- ***Prozesse***: Ein Geschäftsprozess ist eine Folge von geschäftlichen Aktivitäten (= Features) innerhalb eines bestimmten Anwendungsbereiches. Im Rahmen dieser geschäftlichen Aktivitäten wird ein bestimmtes Ergebnis angestrebt und auf die Bedürfnisse, Rechte und Pflichten der beteiligten Personen bzw. Personengruppen eingegangen. An einem Geschäftsprozess sind stets mehrere Personen bzw. Personengruppen beteiligt, ein Feature wird dagegen jeweils nur von einer Person bzw. Personengruppe ausgeführt.

- ***Kommunikation***: Sowohl innerhalb der Geschäftsprozesse als auch zwischen diesen bestehen Kommunikationsstrukturen, welche sich durch die Interaktion der beteiligten Personen bzw. Personengruppen und durch den notwendigen Informationsaustausch ergeben.

Die Analyse und Optimierung der bestehenden Geschäftsprozesse sowie der damit verbundenen Kommunikationsstrukturen im jeweiligen Anwendungsbereich des universitären Alltags machen eine Vereinfachung und Vereinheitlichung der Abläufe möglich.

Die technische Realisierung der konzeptionellen Aspekte, d.h. der Geschäftsprozesse und Kommunikationsstrukturen, im Rahmen einer (web-basierten) Applikation umfasst zwei Teilbereiche:

- ***Dienste***: Die Geschäftsprozesse eines bestimmten Anwendungsbereichs beinhalten eine Menge von Features, wie bereits oben dargestellt. Die technische Umsetzung eines Features wird durch Verwendung von Diensten erreicht, welche bestimmte Teilfunktionalitäten innerhalb des Anwendungbereiches einer Applikation realisieren. Im Rahmen der Realisierung der Dienste müssen die Bedürfnisse, Rechte und Pflichten der Personen (= Benutzer) bzw. Personengruppen (= Benutzergruppen) berücksichtigt werden, die diese Dienste ausführen sollen.

- ***Koordination***: Die für die verschiedenen Benutzergruppen und Bereiche des universitären Alltags realisierten Dienste werden in Form einer Bibliothek über eine zentrale, web-basierte Plattform bereitgestellt. Diese koordiniert die Zusammenarbeit der einzelnen Dienste und ermöglicht außerdem eine flexible Erweiterbarkeit der zugehörigen Applikation.

Eine web-basierte Applikation betrachten wir somit als eine Menge von Diensten zusammen mit einer Schicht, welche die Zusammenarbeit der einzelnen Dienste koordiniert.

Im Rahmen des WIS-Teilprojektes Nr. 6 wird die Analyse, Optimierung und Vereinheitlichung der Geschäftsprozesse am Fachbereich Informatik der Universität Dortmund angestrebt. Insbesondere soll für diese Geschäftsprozesse eine weitgehende Rechnerunterstützung in Form einer web-basierten Applikation realisiert werden.

Folgende Basisanforderungen werden dafür an die Realisierung gestellt:

- Die mittelbare Kommunikation sowie die Interaktion zwischen den an den Geschäftsprozessen beteiligten Personengruppen wird adäquat über elektronische Medien unterstützt.

- Eine zentrale Verwaltung von Benutzern und deren Rechten wird umgesetzt und kommt im Rahmen der web-basierten Applikation zum Einsatz.

- Eine flexibel erweiterbare Bibliothek, welche eine Integrationsunterstützung für ein wachsendes Dienstpotential beinhaltet, ermöglicht das Bereitstellen der entwickelten Dienste.

Das *Lehremanagement an einer Universität* dient dabei als Beispiel eines Anwendungsbereichs. Eine Rechnerunterützung für die bestehenden Geschäftsprozesse innerhalb dieses Anwendungsbereichs wird im Rahmen des TEMPLUS[3] ([91, 28, 27]) Projektes realisiert.

Der Einsatz der durch dieses Projekt bereitgestellten personalisierten, web-basierten TEMPLUS Applikation bewirkt eine Entlastung der Lehrenden von organisatorischen und administrativen Aufgaben, verschafft ihnen dadurch mehr Zeit, sich auf die inhaltlichen Belange der Lehre sowie die persönliche Betreuung von Studierenden zu konzentrieren, und kann so zu einer Verbesserung der Lehresituation allgemein beitragen.

Beispiel 1.1:

Beispiele für Geschäftsprozesse aus dem TEMPLUS Projekt sind die *Anmeldung zu einer Lehrveranstaltung* und die *Einteilung der Übungsgruppen im Rahmen dieser Lehrveranstaltung*. Verschiedene Personengruppen des universitären Alltags (z.B. Dozent, Organisator, Tutor, Student) sind an diesen Geschäftsprozessen beteiligt:
Anmeldung

- Studierende führen die Features 'zu einer Lehrveranstaltung anmelden' (Dienst *registerForCourse*) und 'zu Gruppen anmelden' (Dienst *groupRegistration*) aus, um sich für eine Lehrveranstaltung und deren zugehörige Übungsgruppen anzumelden.
- Dozent und Organisator haben über das Feature 'Teilnehmerliste anzeigen' (Dienst *showParticipantsCourse*) jederzeit Einsicht in die Teilnehmerliste ihrer Lehrveranstaltung.
- Das Feature 'Anmeldestatistik' (Dienst *showGroupRegistrationStatistics*) liefert dem Organisator einen Überblick darüber, wie beliebt die einzelnen Übungstermine bei den Studierenden sind.

Einteilung

- Das Feature 'Gruppeneinteilung initiieren' (Dienst *distribution*) dient dem Organisator dazu, die Einteilung der Übungsgruppen anhand der Anmeldedaten der Studierenden vorzunehmen. Dabei wird eine möglichst optimale Verteilung angestrebt.
- Die Veröffentlichung der Übungsgruppeneinteilung nimmt der Organisator mithilfe des Features 'Gruppeneinteilung veröffentlichen' (Dienst *publishDistribution*) vor.
- Nach Veröffentlichung der Einteilung können Studierende über das Feature 'Mein Stundenplan' (Dienst *showTimetable*) ihren Stundenplan abrufen.
- Auch die Tutoren haben nach Veröffentlichung der Einteilung über das Feature 'Teilnehmerliste einer Gruppe anzeigen' (Dienst *showDistributionGroup*) Zugriff auf die Teilnehmerlisten der von ihnen moderierten Übungsgruppen.

[3]Der Name TEMPLUS steht für TEACHING MANAGEMENT PLATFORM FOR UNIVERSITIES.

Kapitel 2

Rahmenbedingungen und Ziele

Im Rahmen des WIS-Teilprojekts Nr. 6 wird für die Entwicklung einer web-basierten Applikation – wie z.B. der TEMPLUS Applikation – die bestehende METAFrame *Agent Building Center Service Definition* Entwicklungsumgebung (kurz: *ABC-SD*) eingesetzt.

ABC-SD ([75, 76, 74, 73, 83, 77, 79, 82, 78, 9, 47, 56]) ist eine graphische Entwicklungsumgebung für die komponenten-basierte Entwicklung web-basierter Applikationen. Die Spezifikation des Applikationsverhaltens erfolgt dabei in Form eines einzelnen Dienstes, der mittels eines gerichteten Graphen modelliert wird:

- Die Knoten repräsentieren funktionale Einheiten des Anwendungsbereiches.
- Die Kanten beschreiben den Kontrollfluss der Applikation.

Auf diese Weise erhält man eine monolithische, aber in sich strukturierte Modellierung des Applikationsverhaltens. Makros können zur weiteren Strukturierung verwendet werden.

Ein rollen-basierter Software-Entwicklungsprozesses – eingeführt in [11] – koordiniert die arbeitsteilige Realisierung web-basierter Applikationen mit *ABC-SD* in einem Team von Entwicklern.

Zusätzlich stellt das *ABC-SD* viele integrierte Tools zur Verfügung, die im Rahmen dieses Software-Entwicklungsprozesses u.a. dazu dienen, die Applikationslogik bereits zur Entwicklungszeit zu validieren und somit die Zuverlässigkeit der Applikation zu erhöhen.

Wie bereits in [11] detailliert beschrieben, bildet damit das *ABC-SD* als Werkzeug eine gute Ausgangsbasis für die Entwicklung zuverlässiger web-basierter Applikationen.

Teil II greift kurz die wichtigsten Aspekte des *ABC-SD* auf, welche im Rahmen dieser Arbeit für das Verständnis der Zusammenhänge wichtig sind.

Auf das *ABC-SD* sowie die zugehörigen Konzepte stützt sich nun die Realisierung des WIS-Teilprojektes Nr. 6 sowie der in dessen Kontext entstandenen TeMPlUs Applikation ([91, 28, 27]).

Folgende beiden Hauptaspekte werden dabei im Rahmen dieser Arbeit behandelt:

- Ein Framework für die Entwicklung modular aufgebauter, komplexer, zuverlässiger, personalisierter, web-basierter Applikationen mit *ABC-SD* wird entworfen (siehe Abschnitt 2.1).
- Die Praktikabilität und Relevanz des entwickelten Frameworks wird anhand der Applikation TeMPlUs nachgewiesen und mithilfe von begleitenden Beispielen illustriert (siehe Abschnitt 2.2).

2.1 Framework

Wir betrachten eine web-basierte Applikation als eine beliebig umfangreiche, aber endliche Menge von Diensten zusammen mit einer Schicht, welche die Zusammenarbeit der einzelnen Dienste koordiniert. Diese Schicht bezeichnen wir als Meta-Koordinationsebene (siehe Abb. 2.1).

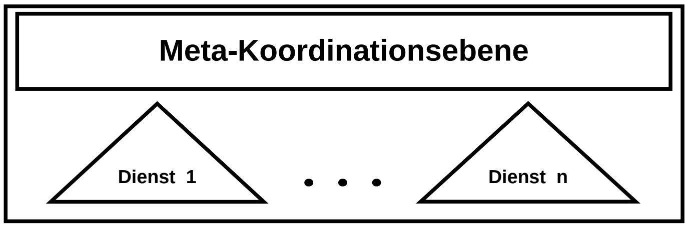


Abbildung 2.1: Modularer Aufbau komplexer web-basierter Applikationen

Die einzelnen Dienste werden dabei unabhängig voneinander als eigenständige Komponenten realisiert, die auf der Datenbank als globalem Zustandsraum arbeiten und über eine Navigation zusammenhängen bzw. ansprechbar sind.

Eine web-basierte Applikation wird dann modular aus diesen Komponenten, den Diensten, zusammengesetzt. Die Meta-Koordinationsebene fungiert hier als Bindeglied, d.h. sie koordiniert die eigenständigen Dienste und verbindet sie damit zu einem Ganzen, der web-basierten Applikation.

Daher bezeichnen wird einen derartig modularen Aufbau einer komplexen web-basierten Applikation auch als 'Dienstfamilie'.

Um diese Aufgabe zu erfüllen, stellt die Meta-Koordinationsebene verschiedene Funktionalitäten zur Verfügung. Wir unterscheiden folgende drei Arten von Funktionalitäten:

- ***Administrationsfunktionalität*** wird vor dem Bereitstellen einer Applikation verwendet, um festzulegen, welche Features die Applikation zur Verfügung stellt.
- ***Verwaltungsfunktionalität*** dient während der Laufzeit einer Applikation zur Verwaltung von Benutzern und deren Rechten.
- ***Navigationsfunktionalität*** bietet dem Benutzer eine strukturierte Navigationsmöglichkeit im öffentlichen und im privaten Bereich einer Applikation an und ermöglicht jeweils den Übergang von einem Bereich in den anderen.

Diese Funktionalitäten werden wiederum selbst mithilfe des *ABC-SD* als Dienste realisiert, die fester Bestandteil des entwickelten Frameworks sind. Wir fassen diese Dienste daher unter dem Begriff 'Basisdienste' zusammen.

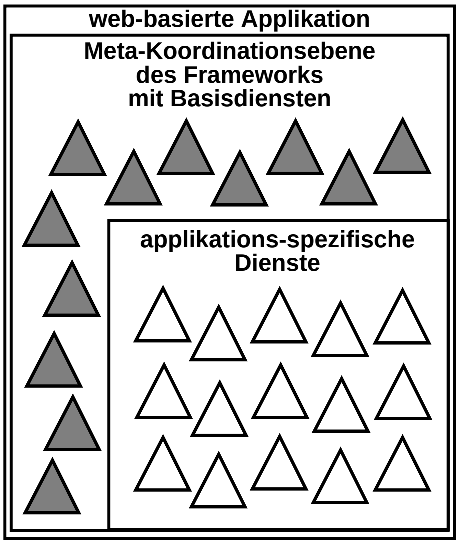


Abbildung 2.2: Framework für web-basierte Applikationen

Abb. 2.2 skizziert das Framework mit den oben beschiebenen 'Basisdiensten' (im Bild als graue Dreiecke dargestellt), welche für beliebige personalisierte, web-basierte Applikationen, die mit *ABC-SD* entwickelt werden, einsetzbar sind. Dagegen sind die übrigen Dienste (im Bild als weiße Dreiecke dargestellt) applikations-spezifisch.

Die Menge der Dienste ist dabei flexibel erweiterbar. Darüber hinaus ermöglicht der modulare Aufbau der Applikation auch ein einfaches Austauschen von Diensten und damit von Features der Applikation.

Bei dem Zusammenspiel der Dienste gibt es i.d.R. keine Schnittstellenproblematik, da alle Dienste die Persistenzschicht der Applikation – d.h. die Datenbank als globalen Zustandraum ansehen und bzgl. dieser innerhalb eines jeden Dienstes geprüft werden kann, ob die für das Ausführen des jeweiligen Dienstes notwendigen Voraussetzungen für den aktuellen Benutzer erfüllt sind. Ist dies nicht der Fall, muss eine entsprechende Fehlerbehandlung innerhalb des Dienstes realisiert werden.

Im Rahmen der Interaktion mit einer personalisierten, web-basierten Applikation stehen zwei verschiedene Bereiche zur Verfügung.

- Ein ***öffentlicher Bereich***, der für jeden Benutzer frei zugänglich ist, stellt über eine öffentliche Navigationsleiste alle öffentlichen Dienste zur Verfügung.

- Im ***privaten Bereich*** stehen dem Benutzer über eine personalisierte Navigationsleiste alle privaten Dienste zur Verfügung, die in ihrem Zugang beschränkt sind und die ein Benutzer nur ausführen darf, wenn er eine entsprechende Berechtigung hierfür besitzt.

Ein Benutzer kann nur in den privaten Bereich einer personalisierten, web-basierten Applikation gelangen, nachdem er sich durch einen *Login-Vorgang* bei der Applikation angemeldet hat. Über einen *Logout-Vorgang* kann sich ein angemeldeter Benutzer wieder abmelden, d.h. er gelangt dann wieder in den öffentlichen Bereich der Applikation.

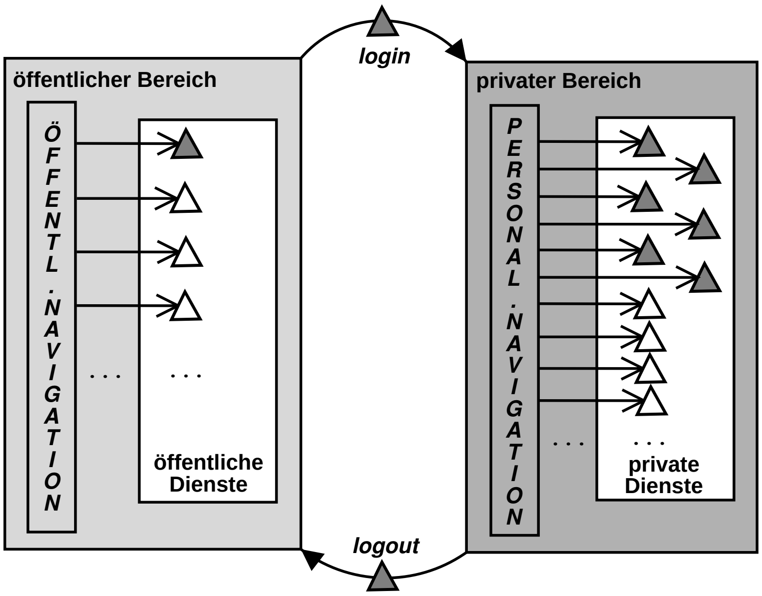


Abbildung 2.3: Framework einer web-basierten Applikation

Abb. 2.3 veranschaulicht diese Zusammenhänge. Dabei unterscheiden wir auch hier in der Darstellung zwischen den Basisdiensten (im Bild als graue Dreiecke dargestellt), welche das Framework bereitstellt, und den applikations-spezifischen Diensten (im Bild als weiße Dreiecke dargestellt).

Durch die Entwicklung einer personalisierten, web-basierten Applikation als Dienstfamilie wird die Komplexität der Applikation beherrschbar. Folgende Aspekte spielen dabei eine entscheidende Rolle im Rahmen der Realisierung des Frameworks für personalisierte, web-basierte Applikationen:

- Die Aufteilung des Gesamtverhaltens einer personalisierten, web-basierten Applikation in einzelne Features, welche wiederum mithilfe von Diensten realisiert werden, entspricht einer ***Dekomposition des Problems*** 'Entwicklung einer personalisierten, web-basierten Applikation für einen bestimmten Anwendungsbereich'.
- Die Dekomposition des Problems ermöglicht eine ***modulare und unabhängige Entwicklung*** der einzelnen Dienste.
- Eine ***Wiederverwendung*** von Basisdiensten aus dem Framework reduziert den Realisierungsaufwand bei der Entwicklung einer personalisierten, web-basierten Applikation.
- Der modulare Aufbau einer personalisierten, web-basierten Applikation aus einer Menge von Diensten erlaubt die ***Konfiguration der Gesamtfunktionalität*** dieser Applikation. Einzelne Features können so einfach, schnell und sicher hinzugefügt, entfernt bzw. ausgetauscht werden.

2.2 Beispiel-Applikation TeMPlUs

Die TeMPlUs Applikation dient als Beispiel-Applikation zum Nachweis der Praktikabilität und Relevanz des entwickelten Frameworks bei der Entwicklung personalsierter, web-basierter Applikationen mit *ABC-SD*. Begleitende Beispiele aus dem TeMPlUs Projekt illustrieren die entwickelten Konzepte im Rahmen dieser Arbeit.

Die TeMPlUs Applikation, welche im Rahmen des WIS-Teilprojektes Nr. 6 entstanden ist, stellt eine Realisierung der im Rahmen der zugehörigen Maßnahmen-Rosette (siehe Abb. 1.1) definierten Anforderungen in Form einer personalisierten, web-basierten Applikation dar. Sie behandelt – wie durch die Abkürzung TeMPlUs (Teaching Management Platform for Universities) auch bereits ausdrückt wird – viele Geschäftprozesse im Rahmen des Lehremanagements am Fachbereich Informatik der Universität Dortmund.

Diese Geschäftsprozesse umfassen den kompletten Lebenszyklus einer Lehrveranstaltung, welcher aus vier Phasen besteht. Jede Phase beinhaltet dabei mehrere Geschäftsprozesse:

- **Planung:** Während der Planungsphase werden beispielsweise die Daten für eine Lehrveranstaltung und deren zugehörige Übungsgruppen festgelegt, Termine zugewiesen, das Lehrpersonal und die Räume zugeteilt, etc.
- **Organisation**: Während der Organisationsphase melden sich Studierende zu Lehrveranstaltungen sowie den zugehörigen Übungsgruppen an, und es erfolgt anschließend die Verteilung von Studierenden auf die einzelnen Übungsgruppen, etc.
- **Durchführung**: Die Durchführung einer Lehrveranstaltung beinhaltet das Bereitstellen von Material und Informationen zur Lehrveranstaltung, den Einsatz von Mailverteilern sowie die Verwaltung übungs-spezifischer Daten, etc.
- **Abschluss**: Den Abschluss einer Lehrveranstaltung bildet schließlich die Erstellung von Scheinen sowie deren Ausgabe an die Studierenden.

Das wesentliche Problem bei der Realisierung besteht darin, die Komplexität der zugrunde liegenden Geschäftsprozesse beherrschbar zu machen. Zusätzlich spielen die große Anzahl an Benutzern mit unterschiedlichen Profilen sowie der Umfang der zu verarbeitenden Datenmengen eine entscheidende Rolle.

Im Einzelnen umfasst das TeMPlUs Projekt zur Zeit ca. 15 Geschäftsprozesse, die sich aus 120 verschiedenen Diensten zusammensetzen. Etwa 2000 Benutzer mit unterschiedlichen Aufgabenbereichen und Bedürfnissen sind bei der Applikation registriert – dabei reicht das Spektrum von den Dozenten und Tutoren bis hin zu den Studierenden. Pro Semester werden im Durchschnitt sechs Lehrveranstaltungen mit bis zu 1000 Teilnehmern inklusive der zu einer Lehrveranstaltung gehörigen, jeweils bis zu 30 Übungsgruppen über das System verwaltet, mit steigender Tendenz. Ein Team von zwischenzeitlich bis zu acht Entwicklern ist bisher an der Realisierung des TeMPlUs Projektes beteiligt gewesen.

Kapitel 3

Designentscheidungen und Meine Anteile

Das TeMPlUs Projekt soll als web-basierte Applikation unter Berücksichtigung folgender wesentlicher Aspekte realisiert werden:

1. Zentral sind Geschäftsprozesse aus dem Bereich Lehremanagement am Fachbereich Informatik der Universität Dortmund (insbesondere aus dem Bereich Organisation und Durchführung von Lehrveranstaltungen an einer deutschen Präsenzuniversität).
2. Die Menge der Funktionalitäten muss über eine flexibel erweiterbare Bibliothek in Form von Diensten zur Verfügung gestellt werden.
3. Da im Rahmen der TeMPlUs Applikation benutzer-spezifische Daten erhoben und gespeichert werden, wird diese als personalisierte, web-basierte Applikation umgesetzt. Wir benötigen demnach eine Verwaltung von Benutzern und deren Rechten.
4. Aufgrund der Komplexität der Applikation ist es besonders wichtig, die Zuverlässigkeit der Applikation überprüfen zu können.

3.1 Funktionale Anforderungen von TeMPlUs

Es gibt verschiedenste web-basierte Tools, die jeweils einen Teil der üblicherweise an einer Universität bestehenden Geschäftsprozesse – wie beispielsweise Anmeldung zu Veranstaltungen und Vergabe von Teilnehmerplätzen, Zuteilung von Lehrpersonal und Räumen, Verwaltung von Materialien, etc. – behandeln können und dabei verschiedene Technologien einsetzen ([100, 97, 23, 86, 95, 96, 53, 29, 30, 13]).

Dabei beschäftigen sich viele dieser Projekte mit dem Bereich des E-Learning, d.h. stellen Online-Kurse bzw. -Lerneinheiten für Fernstudium und/oder Weiterbildung zur Verfügung ([100, 97, 86, 95, 96, 23, 53]). Ein Teil dieser Tools bietet auch Funktionalität im Bereich rechnergestützter Kommunikation an, z.B. Chat, Foren, E-Mail ([97, 53, 100]). [13] unterstützt insbesondere administrative Abläufe, wie die Erstellung eines Vorlesungsverzeichnis und die Hörsaalvergabe. [29] realisiert die Zuteilung von Lehrpersonal zu Veranstaltungen. [30] realisiert eine Überprüfung von Studienleistungen bzgl. der Anforderungen für einen Abschluss an einer amerikanischen Universität verbunden mit einer Studienberatung.

Im Rahmen des TeMPlUs Projektes wollen wir uns jedoch mit Funktionalitäten aus dem Bereich 'Organisation und Durchführung von Lehrveranstaltungen an einer deutschen Präsenzuniversität' befassen, mit dem Hauptziel, Abläufe zu vereinheitlichen und zu vereinfachen sowie die Lehrenden bei administrativen und organisatorischen Tätigkeiten bestmöglich zu unterstützen.

Keines dieser Tools erfüllt jedoch annähernd unsere funktionalen Anforderungen, wie oben dargelegt.

In etwa zeitgleich mit Beginn des TeMPlUs Projektes startete auch das HIS-LSF Projekt an der Universität Dortmund ([31]). Diese Software kommt mit ihrer Menge an Funktionalitäten (Erfassung von Lehrveranstaltungen, Belegen von Veranstaltungen, Unterstützung verschiedener universitärer Nutzerkreise, etc.) TeMPlUs am nächsten, stand aber zu Projektbeginn noch nicht zur Verfügung.

Wir haben uns daher zum damaligen Zeitpunkt entschieden, die personalisierte, web-basierte TeMPlUs Applikation selbst zu realisieren.

Für die drei Bereiche *flexible Erweiterbarkeit*, *Personalisierung* und *Zuverlässigkeit* sollen dabei möglichst allgemein gültige Konzepte entworfen werden, deren Praktikabilität dann im Rahmen der Beispielapplikation TeMPlUs nachgewiesen werden kann.

3.2 Flexible Erweiterbarkeit

Wie bereits in [11] ausführlich dargestellt und mittels einer Abgrenzung gegenüber anderen Komponentenmodellen und Webtechnologien belegt, ist *ABC-SD* das adäquate Tool für die Entwicklung web-basierter Applikationen.

Dabei ermöglichen im Rahmen des *ABC-SD* ein rollen-basierter, sich an den Schichten einer web-basierten Applikation orientierender Software-Entwicklungsprozess, dessen wichtigsten Aspekte nochmals in Kapitel 8 zusammengefasst sind, sowie der Einsatz von Makros die arbeitsteilige Entwicklung der Applikation.

Die Koordinationsschicht, welche die Applikationslogik spezifiziert und bisher mittels eines einzelnen *SLG*s realisiert wird, ist zentral im Rahmen der Entwicklung web-basierter Applikationen mit *ABC-SD*. Die Größe der Koordinationsschicht wird durch die Anzahl der Features der Applikation sowie deren Umfang bestimmt. Die Anzahl der Komponenten, welche für die Realisierung eines Features benötigt werden, spielt also eine entscheidende Rolle.

Makros sind hier hilfreich, indem sie arbeitsteilige Entwicklung und eine Strukturierung der Koordinationsschicht ermöglichen. Andererseits sind Makros nicht eigenständig, sondern können lediglich innerhalb eines Kontextes (anderes Makro bzw. *SLG*) genutzt werden. Das bedeutet insbesondere, dass Makros nicht separat ausführbar oder testbar sind. Es gibt somit einen Verantwortlichen (den Anwendungsexperten), der am Ende der arbeitsteiligen Entwicklung die Applikation zu einem *SLG* zusammensetzen, validieren, generieren und testen muss.

Dies ist bei einer wachsenden Anzahl von Features sehr aufwändig, zunehmend unübersichtlich und nur schwer wartbar. Somit skaliert der Ansatz aus [11] zwar vertikal – d.h. bzgl. der im Entwicklungsprozess festgelegten Rollen – die Zusammenführung der Teilergebnisse, die Validierung sowie das Testen lastet aber auf den Schultern eines einzelnen Verantwortlichen.

Da die Komplexität der web-basierten TEMPLUS Applikation – d.h. die Größe ihrer Koordinationsschicht – flexibel erweiterbar sein soll, muss demnach eine Möglichkeit gefunden werden, diese Komplexität beherrschbar zu machen und die Arbeitsbelastung besser zu verteilen.

Ein Makro stellt eine Komponente einer Applikation dar. Dagegen ist ein Dienst eine Applikation, die auch als Komponente verwendet werden kann.

Indem wir nun die Applikationslogik auf mehrere Dienste (d.h. *SLG*s) aufteilen, erreichen wir eine Modularisierung. Die Anzahl dieser Dienste ist variabel sowie flexibel erweiterbar, und wie in Abschnitt 2.1 dargestellt ergeben sich hierdurch viele Vorteile.

Dienste sind demnach Komponenten ([93, 70, 87, 26]), die Applikationscharakter besitzen:

- klar abgegrenzte Teile der Gesamtfunktionalität einer Applikation, die eine bestimmte Dienstleistung erfüllen
- eigenständig
- klare Schnittstelle
- separat testbar
- separat versionierbar

- einfach wartbar aufgrund geringerer Größe
- arbeitsteilig entwickelbar

Der im Rahmen dieser Arbeit vorgestellte, neue Ansatz beschreibt einen erweiterten, an den modularen Aufbau einer web-basierten Applikation angepassten Entwicklungsprozess (siehe Kapitel 13), der nicht nur vertikal (bzgl. der Rollen im Entwicklungsprozess) skaliert, sondern zusätzlich auch horizontal (bzgl. der Anzahl der Dienste).

Auf Entwicklungsebene wird somit die Arbeitsbelastung beim Validieren, Generieren und Testen der einzelnen Dienste auf mehrere verantwortliche Anwendungsexperten verteilbar, d.h. die Arbeitsteiligkeit wird erhöht.

Für die Laufzeit- bzw. Produktionsumgebung ergibt sich auch ein entscheidender Vorteil. Die Gesamtfunktionalität der Applikation wird konfigurierbar. Dieses sog. Software-Konfigurationsmanagement erlaubt einfaches Hinzufügen, Austauschen, Entfernen von Funktionalität im Rahmen der Applikation.

Dies ist vergleichbar mit dem Bereich der Telekommunikation. Dort gibt es die sog. Features als kleinste für den Benutzer sichtbare und ausführbare funktionale Einheiten, z.B. Rufweiterleitung oder Kurzwahl. Ein Telefon wird dann mit einer Menge (d.h. einem Paket) solcher Features programmiert. Dabei können Features an- oder ausgeschaltet bzw. ersetzt werden.

3.3 Personalisierung

Ein Teil der Gesamtfunktionalität einer personalisierten, web-basierten Applikation wie TeMPlUs bezieht sich auf den Bereich Personalisierung.

> 'Die Idee der komponentenbasierten Softwareentwicklung ist es, komplette Anwendungen unter Verwendung von Bausteinen, die Teile der Anwendungslogik realisieren, zu bauen. Je spezialisierter Komponenten sind, desto geringer ist die Chance für ihre Wiederverwendung. Wenn sie aber dann wiederverwendet werden können, dann stiften sie großen Nutzen.'
> ([93])

Wir wollen demnach ein möglichst allgemeines, wiederverwendbares Konzept für den Bereich der Personalisierung entwickeln und im Rahmen der TeMPlUs Applikation einsetzen.

Der Begriff Personalisierung selbst ist breit gefächert und spielt in immer mehr Bereichen des täglichen Lebens (Online-Shops, Online-Banking, Online-Auktionen, Online-Dienste kommunaler Einrichtungen oder sonstiger Institutionen, Webportale von Firmen für ihre Kunden, etc.) eine wesentliche Rolle.

Eine einheitliche Definition des Begriffs Personalisierung existiert dabei nicht. Man unterscheidet generell folgende drei Bereiche ([66, 69, 61]):

- rollen- und/oder rechte-basiertes System
- kundenspezifische Anpassungen (= customization)
- Analyse und Auswertung des Browsing-Verhaltens der Benutzer (= tracking)

Da die beiden letzteren Aspekte nur applikations-spezifisch realisierbar sind, beschränken wir uns im Rahmen dieser Arbeit auf den ersten Aspekt, die Realisierung eines rollen- und rechte-basierten Systems.

Dies ist vergleichbar mit den Benutzergruppen (*groups*) und Zugriffsrechten (*read*, *write* für Dateien sowie bei ausführbaren Programmen zusätzlich *execute*), die im Rahmen von Betriebssystemen verwendet werden. Dabei verwenden wir im Rahmen eines rollen- und rechte-basierten System zunächst nur Zugriffrechte des Typs *execute*. Diese entsprechen der Erlaubnis ein Feature auszuführen.

Das *ABC-SD* stellt bereits eine vordefinierte Bibliothek von Komponenten (*SIB*s) für den Bereich des Benutzer- und Rollenmanagements ([48]) zur Verfügung.

Allerdings fehlt hier eine Möglichkeit, die Features einer Applikation zu verwalten und sie mit den Benutzern und Rollen zu assoziieren.

Im Rahmen dieser Arbeit wird dementsprechend das Benutzer- und Rollenmanagement des *ABC-SD* um eine Möglichkeit zur Verwaltung der Features einer Applikation erweitert, welche gekoppelt ist an das bestehende Benutzer- und Rollenmanagement.

In Kapitel 11 wird ein Personalisierungsframework präsentiert, das dann unter Verwendung der Komponenten aus [48] eine Benutzer-, Rechte- und Featureverwaltung sowie entsprechende Navigationsfunktionalität als wiederverwendbare Basisdienste realisiert und für web-basierte Applikationen zur Verfügung stellt.

Dabei umfasst der Begriff Personalisierung – wie er im Rahmen dieser Arbeit verwendet wird – folgende Bereiche:

- Anpassung des Applikationsverhaltens sowie der enthaltenen Funktionalitäten an einzelne Benutzer und Benutzergruppen sowie deren Bedürfnisse, Berechtigungen und Verpflichtungen
- Einschränkung und Prüfung der Zugriffsberechtigung anhand der für einen Benutzer gespeicherten Rechte

- Anbieten von Navigationsmöglichkeiten sowie Präsentation von Inhalten für einen Benutzer basierend auf den Daten, die für diesen Benutzer gespeichert sind, d.h. Erzeugen persönlicher Sichten

3.4 Zuverlässigkeit

Darüber hinaus wollen wir uns auch im Bereich Validierung mit einem wichtigen, applikations-unabhängigen Problem befassen, das auch in der OO-Programmierung ([67]) eine wichtige Rolle spielt. Dort wird im Rahmen der Übersetzung eines Programmes überprüft, ob jede verwendete Variable auch zuvor initialisiert worden ist.

Eine vergleichbare Situation liegt bei der Entwicklung web-basierter Applikationen mit *ABC-SD* vor. Dort werden während der Modellierung der Koordinationsschicht einer Applikation, d.h. der *SLG*s, ebenfalls Variablen verwendet, auf denen die einzelnen Komponenten (*SIB*s und Makros) arbeiten.

Die Verwendung nicht initialisierter Variablen kann hierbei zu gravierenden Fehlersituationen beim Einsatz der Applikation führen, wie z.B.

- Absturz eines Dienstes,
- Zugriff auf nicht erlaubte Bereiche oder
- Anzeige von unvollständigen bzw. fehlerhaften Webseiten.

Die Überprüfung, ob jede verwendete Variable initialisiert ist, könnte mittels Datenflussanalyse durchgeführt werden ([60, 62]).

Wie in [71, 72, 68, 57] beschrieben, kann Modelchecking auch zur Datenflussanalyse eingesetzt werden. Das *ABC* stellt bereits einen integrierten Modelchecker zur Verfügung. Die Eigenschaften, für welche wir uns interessieren, können dann temporallogisch ([18, 19, 20]) spezifiziert werden und mithilfe dieses Modelcheckers überprüft werden.

Um das Problem 'Keine Verwendung nicht initialisierter Variablen' zu lösen, wird nun ein sog. Scope-Checker (siehe Kapitel 16) entwickelt. Dieser stellt – unter Einsatz des in das *ABC* integrierten *MCGame*-Modelcheckers ([58, 99]) – für eine beliebige, web-basierte und mit *ABC-SD* erstellte Applikation sicher, dass im Rahmen der Modellierung der Koordinationsschicht ausschließlich initialisierte Variablen verwendet werden.

Dabei muss der Anwendungsexperte sich nicht mit temporallogischen Eigenschaften auseinandersetzen, da diese in Form von vordefinierten Bibliotheken zur Verfügung stehen und für die jeweilige Koordinationsschicht instantiiert werden.

So können bereits während der Validierungsphase viele der oben beschriebenen Fehler frühzeitig identifiziert und behoben werden, was die Zuverlässigkeit der entwickelten Applikation erhöht.

3.5 Meine Anteile

Meine Anteile an den im Rahmen dieser Arbeit vorgestellten Ergebnisse umfassen

- die Entwicklung eines Personalisierungsframeworks mit einer Menge wiederverwendbarer Basisdienste (siehe Kapitel 11 und 12) als Erweiterung eines bestehenden, einfachen Benutzer- und Rollenmanagements,
- die Beschreibung des Entwicklungsprozesses $MyProcess_{ABC\text{-}SD}$, welcher an den modularen Aufbau einer web-basierten Applikation als Dienstfamilie sowie an den Einsatz des entwickelten Personalisierungsframeworks angepasst ist (siehe Kapitel 13), und
- die Realisierung eines Scope-Checkers, welcher die Zuverlässigkeit der entwickelten Dienste erhöht (siehe Kapitel 16), unter Verwendung des in das *ABC* integrierten Modelcheckers.

Kapitel 4

Überblick

Im Rahmen dieser Arbeit sind die Präsentation eines modularisierten ***Frameworks*** für die Entwicklung zuverlässiger, personalisierter, web-basierter Applikationen mit *ABC-SD* von zentraler Bedeutung.

Die Grundlage für dieses Framework bildet der modulare Aufbau einer web-basierten Applikation aus einer Menge von Diensten, der sogenannten ***Dienstfamilie*** (siehe Abschnitt 2.1).

Begleitende ***Beispiele aus dem*** TEMPLUS ***Projekt*** illustrieren dabei jeweils die entwickelten Konzepte.

Zunächst greift Teil II kurz die wichtigsten Aspekte der eingesetzten Entwicklungsumgebung *ABC-SD* auf, welche im Rahmen dieser Arbeit für das Verständnis der Zusammenhänge wichtig sind.

Teil III stellt das ***Personalisierungsframework für web-basierte Applikationen*** vor:

- In Kapitel 9 werden die verschiedenen *Szenarien der Wiederverwendbarkeit von Diensten* dargestellt.
- Kapitel 10 beschreibt eine Klassifikation der Komponenten personalisierter, web-basierter Applikationen mithilfe einer *Taxonomie.*
- Kapitel 11 beschreibt detailliert die *Analyse und Realisierung des Personalisierungsframeworks* und präsentiert einige Anwendungsbeispiele.
- Kapitel 12 zeigt in Form eines *Leitfadens*, welche Vorbereitungen für den Einsatz des eingeführten Frameworks vorzunehmen sind.

Der modulare Aufbau einer personalisierten, web-basierten Applikation als Dienstfamilie macht außerdem eine Anpassung bzw. Erweiterung des bisher im Rahmen des *ABC-SD* verwendeten Software-Entwicklungsprozesses ([11]) sowie der integrierten Validierungsmethoden notwendig. Im Folgenden wird zur Referenzierung dieses Software-Entwicklungsprozesses die Abkürzung $PM_{ABC\text{-}SD}$ verwendet.

In Teil IV wird erklärt, wie sich der ***Software-Entwicklungsprozess*** $PM_{ABC\text{-}SD}$ in einen ***koordinativen Prozessrahmen einbetten*** lässt, der den Einsatz des Personalisierungs-Frameworks adäquat unterstützt. Zusätzlich erfolgt hier eine ausführliche Analyse der bestehenden Schnittstellen und Abhängigkeiten, welche sich aus der rollen-basierten, arbeitsteiligen Entwicklung personalisierter, web-basierter und als Dienstfamilie konzipierter Applikationen mit *ABC-SD* ergeben.

Teil V beschäftigt sich mit den ***Auswirkungen und neuen Möglichkeiten***, die sich im Rahmen der Entwicklung einer web-basierten Applikation als Dienstfamilie ***für die Validierungsmechanismen*** im Rahmen des *ABC-SD* ergeben:

- In Kapitel 14 wird einigen zuvor analysierten Schnittstellen und Abhängigkeiten jeweils eine adäquate Validierungsmethode des *ABC-SD* zugeordnet.

- Kapitel 15 stellt einen Katalog von *Eigenschaften* einer personalisierten, web-basierten Applikation vor, welche *lokal überprüft* werden können.

- Kapitel 16 enthält einen Katalog von *globalen Eigenschaften*, die für personalisierte und mit dem *ABC-SD* erstellte, web-basierte Applikationen *durch Modelchecking überprüft* werden können. Diese globalen Eigenschaften definieren Regeln *für den Applikations- sowie für den Dienstaufbau.*

  Der Schwerpunkt liegt in diesem Kapitel auf der Beschreibung von Regeln für einen sogenannten ***Scope-Checker***, welcher sicherstellt, dass alle in den verschiedenen Datenkontexten (=Scopes) des *ABC-SD* verwendeten Variablen auch zuvor initialisiert worden sind.

  Zentral bei der Überprüfung dieser Eigenschaften mittels Modelchecking ist ein ***Präprozess***, welcher auf Basis der einer Applikation zugrunde liegenden Taxonomie das Modell erzeugt, das die Eingabe für den Modelchecker bildet, und eine gewählte globale temporallogische Eigenschaft für das konkrete Modell initialisiert.

Teil VI enthält mit Kapitel 17 eine kurze ***Zusammenfassung der präsentierten Ergebnisse*** sowie einen Ausblick auf einige ***mögliche Erweiterungen des vorgestellten Frameworks*** für personalisierte, web-basierte Applikationen.

Teil II

Web-basierte Applikationen mit *ABC-SD*

Kapitel 5

Die *ABC-SD* Entwicklungsumgebung

MetaFrame *Agent Building Center* ([75, 74, 56]) – kurz *ABC* – ist eine generische, graphische und modul-basierte Entwicklungsumgebung für eine anwendungsspezifische, komponenten-basierte Softwareentwicklung.

Bei einer topologischen Betrachtung des *ABC* ergibt sich ein 3-Schichten-Modell, das Darstellungsebene, Koordinationsebene und Modulebene unterscheidet (siehe Abb. 5.1). Die mittlere Schicht bildet dabei ein flexibler Interpreter, der auf einer einfachen, PASCAL-ähnlichen, typisierten, imperativen Programmiersprache namens *High-Level Language* (kurz *HLL*) arbeitet [33, 14, 37]. Zusätzliche Typen und Funktionalitäten können in sogenannten Modulen definiert und dynamisch im Interpreter bereitgestellt werden [40]. Der Interpreter fungiert als Kontroll- und Koordinations-Instanz für die verwendeten Module, d.h. er regelt sowohl die Kommunikation zwischen den Modulen untereinander als auch zwischen den Modulen und der graphischen Benutzeroberfläche, und bildet damit den zentralen Kern der Entwicklungsumgebung.

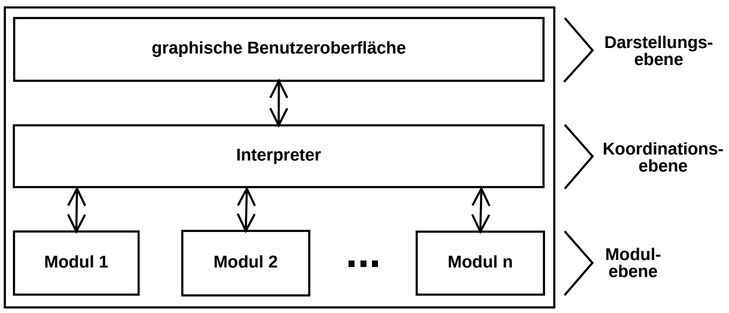


Abbildung 5.1: Topologische Sicht auf das *ABC*

Zwei zentrale vordefinierte Module stellen die Graphformate zur Verfügung, welche die Basis für die Software-Entwicklung mit dem *ABC* bilden:

- Das Modul *PLGraph* [34] des *ABC* definiert eine flexible Datenstruktur für Graphen (*Polymorphic Labelled Graph*s). Das Hauptmerkmal dieser Graphen ist, dass alle Bestandteile eines Graphen (d.h. der Graph selbst, seine Knoten und seine Kanten) Informationen in Form von beliebig vielen sog. Labels speichern können.

- Das Modul *SD* [88, 56] ist eine wichtige Erweiterung des *PLGraph* Moduls, das ein spezialisiertes Graphformat und eine graphische Benutzeroberfläche für die Entwicklung komplexer, workflow-basierter Applikationen bereitstellt.

Das *ABC* wurde bereits in verschiedenen Bereichen für die Entwicklung komplexer, workflow-basierter Applikationen erfolgreich eingesetzt. Dabei gibt es für jeden dieser Bereiche eigene Entwicklungsumgebungen, die auf dem *ABC* und seinen Konzepten basieren und dieses anwendungsspezifisch erweitern.

- Telefonie-Applikationen wurden im Rahmen des *Intelligent Network (IN)* Projektes mithilfe der auf dem *ABC* basierenden MetaFrame *Agent Building Center Service Definition* Entwicklungsumgebung – kurz *ABC-SD* – entwickelt [75, 10].

- Das *ABC* bildet die Basis für die *Electronic Tool Integration (ETI)* Plattform [11]. Diese dient als elektronisches Kommunikationsmedium für Anbieter von Tools als auch für Endbenutzer, die auf der Suche nach Tools sind, mithilfe derer sie ihre Probleme lösen können.

- Eine zentrale Rolle spielt das *ABC* im *Integrated Test Environment (ITE)* [63] bei der Entwicklung und Verifikation von Testfällen sowie der Durchführung der Tests.

- In [47] wird ein personalisierter Internetdienst für wissenschaftliche Begutachtungsprozesse beschrieben (*Online Conference Service*, kurz *OCS*). Dieser personalisierte Internetdienst wurde mithilfe der auf dem *ABC* basierenden *ABC-SD* Entwicklungsumgebung realisiert.

Mithilfe der *ABC-SD* Entwicklungsumgebung – einer Erweiterung des *ABC* – kann eine web-basierte Applikation als sog. *Enhanced Web Information Service* (*EWIS*) realisiert werden. Die grundlegenden Begriffe, welche im Rahmen des *ABC-SD* verwendet werden, sind in Abschnitt 5.1 kurz zusammengestellt.

Das Applikationsverhalten als gerichteter Graph (*Service Logic Graph*, kurz *SLG*) modelliert. Dieser besteht aus Knoten, den funktionalen Einheiten (sog. *Service Independent Building Blocks*, kurz *SIB*s), und gerichteten Kanten, die den Kontrollfluss repräsentieren.

Die graphische Konfiguration der Workflows einer Applikation in Form der *SLG*s liefert die nötige Flexibilität, um diese schnell an mögliche Änderungen anpassen zu können. Makros machen die komplexen Workflows beherrschbar und unterstützen die Aufgabenteilung während der Entwicklung von Applikationen.

Eine Trennung der Daten (Implementierung der *SIB*s) von dem konkreten Workflow ermöglicht Aufgabenteilung und Spezialisierung während der Dienstentwicklung. Ein Software-Entwicklungsprozess für web-basierte Applikationen koordiniert dabei die beteiligten Personen in allen Phasen der Software-Entwicklung.

Zusätzliche vordefinierte Module innerhalb des *ABC* erhöhen die Zuverlässigkeit der entwickelten Applikationen und vereinfachen deren Entwicklung:

- Die Workflow-Validierung (siehe Abschnitt 5.2) erfolgt durch
  - Überprüfung lokaler Bedingungen (Modul *SDLocalCheck*),
  - Überprüfung globaler Bedingungen (Modul *McGame*),
  - symbolische Ausführung (Module *Tracer* und *SDTracer*)
- Das Modul *SDWISCompilation* ermöglicht die automatische Generierung der Applikation, d.h. der Übergang von der Spezifikation der Applikationsslogik (*SLG*) zur Implementierung als Servlet (konkreter, ausführbarer Dienst) wird durch einen automatischen Übersetzungsprozess realisiert (siehe Abschnitt 5.3).
- Der Einsatz vorgefertigter Bibliotheken von Komponenten (*SIB*s) für bestimmte Aufgabenbereiche, z.B. für die Entwicklung web-basierter Applikationen (siehe Abschnitt 5.4), wird durch die Projektverwaltung des *ABC* (Modul *ProjectMgmt*) ermöglicht.

5.1 Grundlegende Begriffe

In den folgenden Abschnitten werden einige grundlegende Begriffe für den Einsatz der *ABC-SD* Umgebung bei der Entwicklung web-basierter Applikationen eingeführt.

Service Logic Graph (*SLG*): Ein *Service Logic Graph* – kurz *SLG* – ist ein gerichteter Graph, dessen Knoten *SIB*s oder Makros sind und dessen Kanten den Kontrollfluss der Applikation repräsentieren (siehe Abb. 5.2, linkes Fenster). Die Applikationslogik, die die Menge der möglichen Workflows innerhalb der Applikation und damit deren Koordinationsschicht (siehe Abschnitt 6) darstellt, wird als *SLG* bzw. Menge von *SLG*s modelliert.

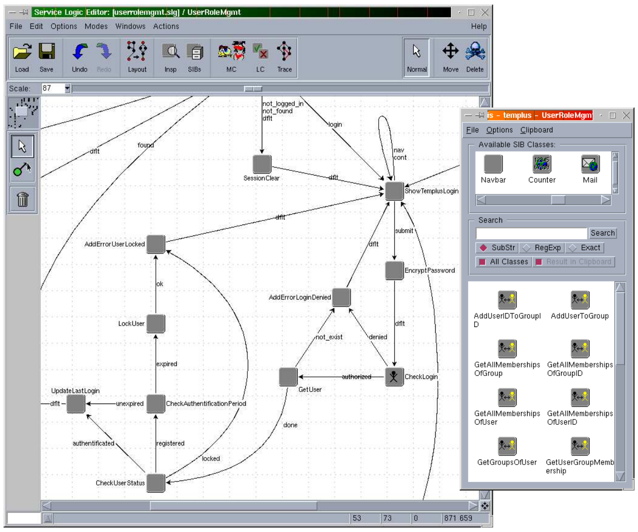


Abbildung 5.2: Graphische Konfiguration von Workflows mit *ABC-SD*

Service-Independent Building Block (*SIB*): Die funktionalen Einheiten oder Komponenten aus denen ein *SLG* zusammengesetzt werden kann, bezeichnen wir als *Service Independent Building Blocks* – kurz *SIB*s. Alle für die Spezifikation der Applikation zur Verfügung stehenden *SIB*s werden in der *SIB*-Palette angeboten (siehe Abb. 5.2, rechtes Fenster) und können dort ausgewählt werden. Alle *SIB*s bestehen aus einer Schnittstellendefinition, welche die Parameter sowie die möglichen ausgehenden Kanten (=Branches) der Komponente festlegt, und einer zugehörigen *JAVA*-Implementierung ([1, 22]).

Makro: Ein Makro ist ein Teilgraph, der als neue Komponente der Applikation gespeichert wird (siehe Abb. 5.3). Jedes Makro hat – wie ein *SIB* – eine Schnittstellendefinition, welche seine Parameter und Branches beschreibt. Über die *SIB*-Palette werden alle für die Spezifikation der Applikation zur Verfügung stehenden Makros angeboten (siehe Abb. 5.2, rechtes Fenster) und können dort ausgewählt werden. Sie sind durch ein kleines `M` an der rechten oberen Ecke des Knotens von den normalen *SIB*s zu unterscheiden. An die Stelle der Implementierung rückt bei Makros der zugehörige Teilgraph als Realisierung. Makros können andere Makros enthalten, dabei dürfen jedoch keinerlei zyklische Abhängigkeiten bestehen.

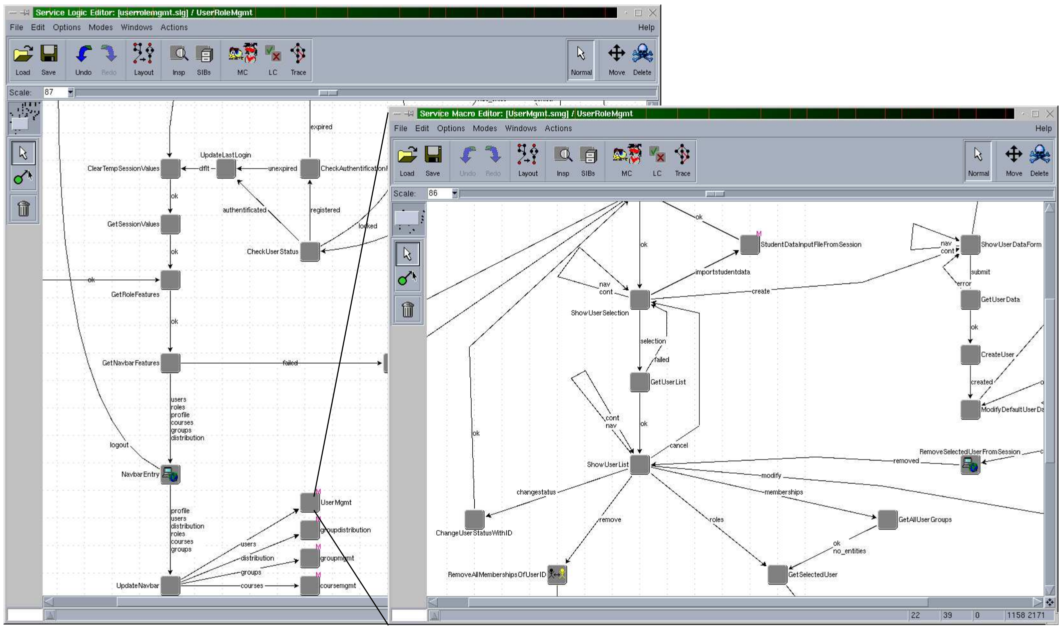


Abbildung 5.3: Makros im *ABC-SD*

Projekt: Ein *Projekt* innerhalb des *ABC* ([12]) kann auf beliebig vielen anderen dort verfügbaren Projekten desselben Typs basieren, deren Komponenten, d.h. *SIB*s und Makros, dann in dem neuen Projekt auch über die *SIB*-Palette zur Verfügung stehen. Darüber hinaus können in dem neuen Projekt beliebig viele eigene *SIB*s, Makros und *SLG*s definiert werden. Beispielsweise sind web-basierte Applikationen vom Projekttyp `EWIS`.

Die Projektdefinition innerhalb der *ABC* erfolgt dabei durch eine XML-Datei mit Endung `.mfp`. Neben dem Namen des Projektes wird hier auch die Liste derjenigen Projekte spezifiziert, auf denen das neue Projekt direkt basiert. Transitive Projektabhängigkeiten werden vom *ABC* automatisch berücksichtigt.

Abb. 5.4 zeigt als Beispiel die `.mfp` Datei für das PwisBase Projekt (siehe Abschnitt 11.2.3). Dabei gibt der Projekttyp `EWIS` an, dass es sich um ein Projekt für die Entwicklung einer web-basierten Applikation handelt. Das PwisBase Projekt basiert auf dem Projekt `Base`, das alle grundlegenden *SIB*s für die Entwicklung web-basierter Applikationen enthält, sowie den Komponenten-Bibliotheken, die durch die Projekte `Database`, `User` und `Communication` definiert werden und jeweils spezialisierte *SIB*s für die entsprechenden Teilbereiche zur Verfügung stellen (siehe Abschnitt 5.4).

```
<?xml version="1.0" encoding="iso-8859-1"?>
<std-project>
  <name>PWISBase</name>
  <type>EWIS</type>
  <included-projects>
    <project>Base</project>
    <project>Database</project>
    <project>User</project>
    <project>Communication</project>
  </included-projects>
</std-project>
```

Abbildung 5.4: Definition des PWISBase Projekts – `PWISBase.mfp`

Applikationslogik: Der Begriff *Applikationslogik* im Kontext web-basierter Applikationen steht für die Spezifikation des Verhaltens bzw. des Kontrollflusses der web-basierten Applikation, d.h. der Interaktionspunkte sowie der internen Abläufe, die für die Beantwortung von Benutzeranfragen oder für die Vorbereitung der Anzeige von Daten notwendig sind. Innerhalb des *ABC-SD* wird die Logik eines web-basierten Dienstes mittels eines *SLG*s spezifiziert.

Dienst: Der Begriff *Dienst* bezeichnet den ausführbaren Teil einer web-basierten Applikation, dessen Ablaufverhalten durch einen *SLG* modelliert und mithilfe der automatischen Code-Generierung (siehe Abschnitt 5.3) aus dieser Spezifikation erzeugt wird. Technisch betrachtet wird ein Dienst durch ein Servlet realisiert. Ein Dienst kann durch Eingabe der zugehörigen URL in den Browser ausgeführt werden. Dabei enthält die URL folgenden Bestandteile:

```
http[s]://<Server>[:<Port>]/<ServletContainer>/servlet/<ServiceName>
```

- Das verwendete Protokoll, also hier http bzw. https bei web-basierten Applikationen die SSL verwenden.
- Der vollständige Domainname (z.B. templus.cs.uni-dortmund.de) oder die IP-Adresse (z.B. 129.217.29.65) des Servers, der die Applikation bereitstellt, wird über `<Server>` angegeben.
- Abhängig von der jeweiligen Applikation kann optional ein `Port` angegeben werden, ansonsten wird der Standardport 8080 verwendet.
- `<ServletContainer>` gibt den Namen des Verzeichnisses – des sog. Servlet-Containers – an, in dem die Applikation auf dem Server installiert ist.
- `<ServiceName>` ist dabei der Name des *SLG*s, durch den der jeweilige Dienst spezifiziert wird.

Workflow: Ein *Workflow* ist eine beliebig lange, zusammenhängende Folge von Interaktionen eines Benutzers mit einer web-basierten Applikation gesteuert durch Mausklicks durch den Benutzer – u.U. auch gekoppelt an die Eingabe von Daten in Formulare – zwischen denen beliebig viele interne Aktionen stattfinden können. Technisch gesehen wird jeder Workflow realisiert durch einen entsprechenden Pfad innerhalb der Applikationslogik.

5.2 Validierung

Das *ABC* Tool bietet verschiedene Mechanismen ([80, 78, 76, 9]) an, die es erlauben, Fehler im Aufbau der Applikationslogik bereits zur Entwicklungszeit aufzudecken, also bevor mittels automatischer Codegenerierung (siehe Abschnitt 5.3) die Implementierung für die Applikationslogik erzeugt wird und die Testphase beginnt. Folglich können dadurch viele Fehler bereits in einer sehr frühen Phase des Softwareentwicklungsprozesses gefunden und behoben werden. Dies reduziert die Kosten und die Zeit bis zur Fertigstellung und Inbetriebnahme der web-basierten Applikation.

Überprüfung Lokaler Eigenschaften: Lokale Eigenschaften beziehen sich immer auf die lokale Konsistenz eines einzelnen *SIB*s (z.B. ob die Parameter und ausgehenden Kanten eines *SIB*s richtig konfiguriert sind) und werden als sogenannter *Local Check Code* (*LCC*) in der Interpreter-Sprache *HLL* ([37]) des *ABC* spezifiziert.

Wenn ein Benutzer bei der Verwendung und Konfiguration dieses *SIB*s einen Fehler macht, wird bei der Durchführung der lokalen Überprüfung innerhalb des *ABC* eine entsprechende Meldung angezeigt, die die Art des Fehlers beschreibt und zusätzliche Hinweise zum Beheben des Fehlers enthalten kann. Die farbige Markierung der Fehlerstelle im *SLG* ermöglicht es dem Benutzer dabei, den Fehler schnell zu lokalisieren.

Details zum Verfassen von *Local Check Code* sind in [38] beschrieben. Abschnitt 15 definiert und illustriert eine Menge grundlegender lokaler Eigenschaften, die bei der Entwicklung personalisierter, web-basierter Applikationen mit *ABC-SD* eingehalten werden müssen.

Beziehungen zwischen verschiedenen *SIB*s können nicht als lokale Eigenschaften spezifiziert werden, sondern müssen als globale Eigenschaften beschrieben werden.

Überprüfung Globaler Eigenschaften: Im Gegensatz zu den lokalen Eigenschaften, bei der jeweils ein einzelner *SIB* auf seine Konsistenz geprüft wird, beschreiben globale Eigenschaften Beziehungen zwischen verschiedenen *SIB*s und beziehen sich damit auf die Graphstruktur. Durch die Überprüfung globaler Eigenschaften mittels Modelchecking [15, 94, 54, 59, 81, 52, 17]) kann demnach die globale Konsistenz eines *SLG*s verifiziert werden.

Für die Überprüfung der Gültigkeit temporaler Bedingungen im Modell kann der über das Modul *MCGame* integrierte Modelchecker ([58, 99]) verwendet werden. Dieser benutzt als Eingabesprache den modalen Mu-Kalkül ([7, 44, 6]) mit Rückwärtsmodalität, stellt darüber hinaus aber auch die bekannten *CTL*-Operatoren ([16]) sowie eine Rückwärts-Variante dieser *CTL*-Operatoren als Makrokonstrukte der Eingabesprache zur Verfügung.

In Kapitel 16 wird der Einsatz des *MCGame*-Modelcheckers bei der Validierung globaler Eigenschaften für personalisierte, web-basierte Applikationen illustriert.

Symbolische Ausführung: Auf Basis der vom *ABC* zur Verfügung gestellten, vordefinierten Module *Tracer* ([36]) und *SDTracer* [39] kann für web-basierte Applikationen ein Simulationsmodul entwickelt werden. Dieses ermöglicht die symbolische Ausführung einer web-basierten Applikation zur Entwicklungszeit auf Ebene des durch den *SLG* spezifizierten Kontrollflusses der Appplikation. Dadurch kann der Entwickler das Laufzeitverhalten der Applikation der zu entwickelnden Applikation während des Spezifizierens des Applikationsverhaltens nachempfinden und so einen ersten Eindruck des Gesamtsystems gewinnen, bevor die Implementierung aller Komponenten fertiggestellt sein muss und ohne sich mit einer konkreten Laufzeitumgebung auseinandersetzen zu müssen.

In [65] wird eine prototypische Implementierung eines solchen Simulationsmoduls für web-basierte Applikationen vorgestellt. Dieses ist als Modul *Sim* in das *ABC* integrierbar.

5.3 Automatische Code-Generierung

Die automatische Code-Generierung wird durch das vordefinierte Modul *SDWIS-Compilation* im *ABC* bereitgestellt. Für einen *SLG* (d.h. Dienst) kann die zugehörige *JAVA*-Implementierung ([1, 22]) erzeugt werden, indem für ihn die automatische Code-Generierung durchgeführt wird.

Jeder *SLG* hat einen Namen, nämlich den Namen des durch diesen *SLG* spezifizierten Dienstes, der im Folgenden mit <service_name> bezeichnet wird. Bei der Durchführung der automatischen Code-Generierung werden dann für einen Dienst <service_name> entsprechende *JAVA*-Dateien generiert und in einem ausgezeichneten Verzeichnis gespeichert, welches alle automatisch generierten Dienstimplementierungen des zu der jeweiligen Applikation gehörigen *ABC* Projektes enthält.

Der Vorteil der Code-Generierung liegt klar darin, dass beim Übergang von der Spezifikation (*SLG*) zur Implementierung (*JAVA*) keine zusätzlichen Fehlerquellen entstehen, da diese Code-Generierung vollautomatisch abläuft. Nach der Compilierung der auf diese Weise generierten *JAVA*-Dateien steht dann ein *JAVA*-Servlet zur Verfügung, das auf einem Server installiert und zur Benutzung freigegeben werden kann.

5.4 Bibliotheken für Web-basierte Applikationen

Die *ABC-SD* Umgebung stellt das Projekt *EWIS* Base zur Verfügung [88], in dem einige grundlegende Komponenten für die Entwicklung web-basierter Applikationen definiert sind.

Darüber hinaus gibt es einige Erweiterungen, die bestimmte Teilbereiche im Rahmen der Entwicklung web-basierter Applikationen abdecken, d.h. Bibliotheken von Komponenten für web-basierte Applikationen zur Verfügung stellen.

Abb. 5.5 gibt einen Überblick über die im Rahmen des *ABC-SD* vordefinierten Komponenten-Bibliotheken sowie die Abhängigkeiten zwischen ihnen.

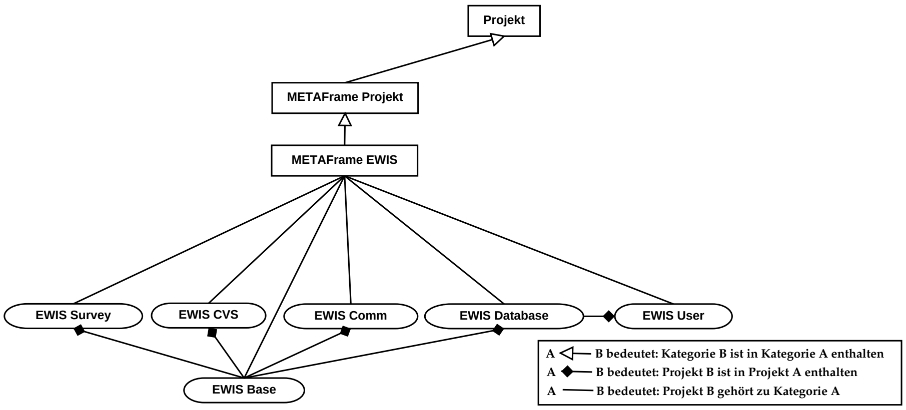


Abbildung 5.5: Komponenten-Bibliotheken des *ABC-SD*

Basiskomponenten Das Projekt *EWIS* Base stellt eine Bibliothek von grundlegenden *SIB*s, d.h. Basiskomponenten, für die Entwicklung web-basierter Applikationen zur Verfügung. In [12] werden diese *SIB*s, ihre Funktionsweise sowie der richtige Umgang mit ihnen beschrieben.

Datenbank-Schnittstelle Das Projekt *EWIS* Database (siehe [3]) definiert eine Bibliothek, welche Komponenten zum Aufbauen und Schließen einer Datenbankverbindung, zum Konfigurieren und Initialisieren von Administratoren, die für jegliche Art von Datenbankmodifikation benötigt werden, sowie Komponenten für Datenbankanfragen und -modifkation beinhaltet.

Benutzer- und Rollen-Management Das Projekt *EWIS* User ([48]) stellt eine Bibliothek zur Verfügung, welche einerseits Komponenten für die Ausführung von Datenbankoperationen beinhaltet, und andererseits Komponenten für ein Benutzer- und Rollenmanagement (wie in [47] beschrieben) bereitstellt.

Mail und News Das Projekt *EWIS* Comm ([42]) stellt eine Bibliothek für die Unterstützung von Mail und News zur Verfügung. Diese beinhaltet Komponenten, mit denen web-basierte Applikationen entwickelt werden können, die die Funktionalitäten einer E-Mail-Applikation oder eines News-Client anbieten, genauergesagt das Senden und Lesen sowie das Bereitstellen und Beantworten von Nachrichten ermöglichen.

Feedback Das Projekt *EWIS* Survey ([35]) stellt eine Bibliothek für das Bereitstellen, Bearbeiten und Auswerten von Online-Fragebögen zur Verfügung. Diese Bibliothek wurde am Lehrstuhl Informatik 5 der Universität Dortmund erweitert. Dabei wurde das in Kapitel 11 vorgestellte Personalisierungsframework eingesetzt sowie eine graphische Aufbereitung der Auswertungsmöglichkeiten ([64]) entwickelt.

CVS Das Projekt *EWIS* CVS stellt eine Bibliothek für die Benutzung von cvs-Funktionalitäten zur Verfügung. Diese beinhaltet Komponenten, mit denen der Inhalt von Repositories modifiziert werden kann, d.h. beispielsweise *SIB*s zum Einchecken und Auschecken von Dateien.

Kapitel 6

Schichten einer Web-basierten Applikation mit *ABC-SD*

Web-basierte Applikationen unterscheiden sich in der Komplexität und technischen Realsierung der Strukturen, die dazu dienen die Anfragen der Benutzer zu bearbeiten und zu beantworten. Wir unterscheiden verschiedene Schichten (siehe Abb. 6.1) für die verschiedenen Aufgaben, die zur Beantwortung einer Anfrage eines Benutzers an eine web-basierte Applikation zu erfüllen sind.

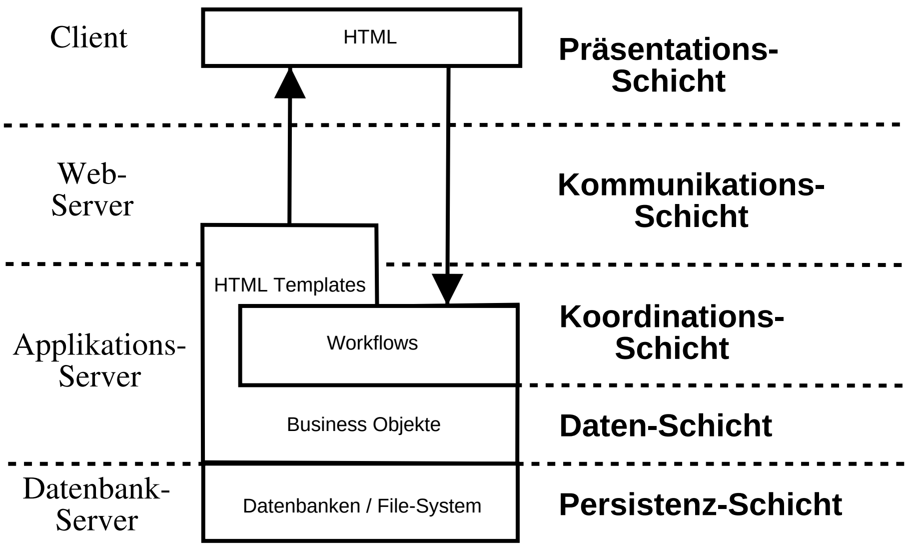


Abbildung 6.1: Die Schichten einer Web-basierten Applikation

In den folgenden Abschnitten werden die einzelnen Schichten, ihre Aufgaben und Besonderheiten kurz erläutert.

Die Präsentations-Schicht bildet die graphische Benutzerschnittstelle einer web-basierten Applikation.

Benutzer interagieren mit einer web-basierten Applikation über einen Browser auf ihrem Client-Rechner. Die HTML-Seiten, die im Browser angezeigt werden, stellen die graphische Benutzeroberfläche (GUI) der Applikation dar und dienen dem Datenaustausch mit der Applikation sowie der Steuerung ihres Verhaltens durch den Benutzer. Zum einen werden Daten und Informationen, die eine web-basierte Applikation bereitstellt, aufbereitet und dem Benutzer über eine HTML-Seite präsentiert oder zum Download angeboten. Zum anderen können Daten und Informationen (durch Formulare, Links etc.) von einem Benutzer abgefragt sowie an die Applikation weitergeleitet und dort verarbeitet oder zur Steuerung des weiteren Kontrollflusses verwendet werden.

Die Kommunikations-Schicht dient der Kommunikation zwischen Benutzer und web-basierter Applikation. Diese beinhaltet zwei grundlegende Aspekte:

- Das Senden von Benutzeranfragen zu dem Server, der die Applikation bereitstellt, über Links und Formulare auf den HTML-Seiten. Dabei können über die mit einem Link assoziierten Parameter bzw. über Textfelder o.ä. eines Formulars auch Daten an die Applikation übermittelt werden.

- Die Beantwortung von Benutzeranfragen durch den Server durch das Anzeigen von neuen HTML-Seiten oder Download-Optionen im Browser des Benutzers. Um diese zu erstellen werden die durch die Applikationslogik definierten internen Aktionen auf dem Server ausgeführt, die zur Bearbeitung der Benutzeranfrage nötig sind.

Die Kommunikations-Schicht dient als Verbindungsglied zwischen der Koordinations-Schicht und der Präsentations-Schicht und befindet sich auf dem Web-Server. In ihr befinden sich die Templates der zu der Applikation gehörigen HTML-Seiten. Diese Templates enthalten Variablen, mithilfe derer auf die von der Koordinations-Schicht über den *Show-Kontext* (siehe Kapitel 7) zur Verfügung gestellten Daten zugegriffen werden kann. Die konkreten HTML-Seiten werden zur Laufzeit aus diesen Templates und den über den *Show-Kontext* verfügbaren Daten generiert. Dieser Prozess läuft auf dem Webserver ab, der sowohl den Kommunikationsweg von der Applikation zum Client durch Generieren der konkreten HTML-Seiten aus den Templates und anschließende Weiterleitung dieser realisiert, als auch die umgekehrte Richtung, wodurch Daten aus Formularen oder Links vom Client zur Applikation gelangen, um dort weiterverarbeitet zu werden.

Die Koordinations-Schicht enthält die Spezifikation und Implementierung aller möglichen Workflows, die eine web-basierte Applikation zur Verfügung stellt, und befindet sich auf dem Applikations-Server. In dieser Schicht wird die konkrete Applikationslogik festgelegt, d.h. der Kontrollfluss der web-basierten Applikation.

Während der Entwicklung der Koordinations-Schicht einer web-basierten Applikation mit *ABC-SD* (siehe Kapitel 5) können verschiedene Datenkontexte zur Speicherung, Zwischenspeicherung und Modifikation von Daten und Geschäftsobjekten verwendet werden, um die Applikationslogik (d.h. die Workflows) zu spezifizieren bzw. zu realisieren. In Kapitel 7 werden diese Datenkontexte, ihre wichtigsten Eigenschaften, sowie der richtige Umgang mit ihnen erläutert.

Die Daten-Schicht beinhaltet die Spezifikation und die Implementierung der Geschäftsobjekte des Anwendungsbereiches, die innerhalb der web-basierten Applikation verwendet werden, und befindet sich auf dem Applikations-Server. Diese Daten können innerhalb der Koordinations-Schicht, d.h. bei der Spezifikation des Kontrollflußes der Applikation verwendet werden. Alle persistenten Geschäftsobjekte des Anwendungsbereiches können in der Persistenz-Schicht der Applikation gespeichert und wieder von dort geladen werden. Diese Abläufe werden mithilfe sog. Datenbank-Administratoren realisiert. Dabei verwaltet ein Datenbank-Administrator Geschäftsobjekte eines speziellen Typs und stellt entsprechende Funktionalitäten zur Verfügung, die Datenbankanfragen und -aktualisierungen aus der Koordinationsschicht der Applikation heraus ermöglichen.

Die Persistenz-Schicht einer Applikation umfasst die Datenbanken und das Filesystem, welche zur Speicherung der verwendeten Geschäftsobjekte dienen, und befindet sich auf dem Datenbank-Server. Der Zugriff auf diese Schicht erfolgt ausschließlich über die sog. Datenbank-Administratoren der Applikation, von denen jeder Geschäftsobjekte eines speziellen Typs verwaltet und entsprechende Funktionalitäten zur Verfügung stellt, die das Ergebnis von Datenbankanfragen zurückliefern und innerhalb der Koordinationsschicht der Applikation verfügbar machen.

Kapitel 7

Datenkontexte einer Web-basierten Applikation mit *ABC-SD*

Im Rahmen der Entwicklung der *Koordinations-Schicht* (siehe Kapitel 6) einer web-basierten Applikation mit *ABC-SD* (siehe Kapitel 5) können verschiedene Datenkontexte verwendet werden – wie in Abb. 7.1 dargestellt.

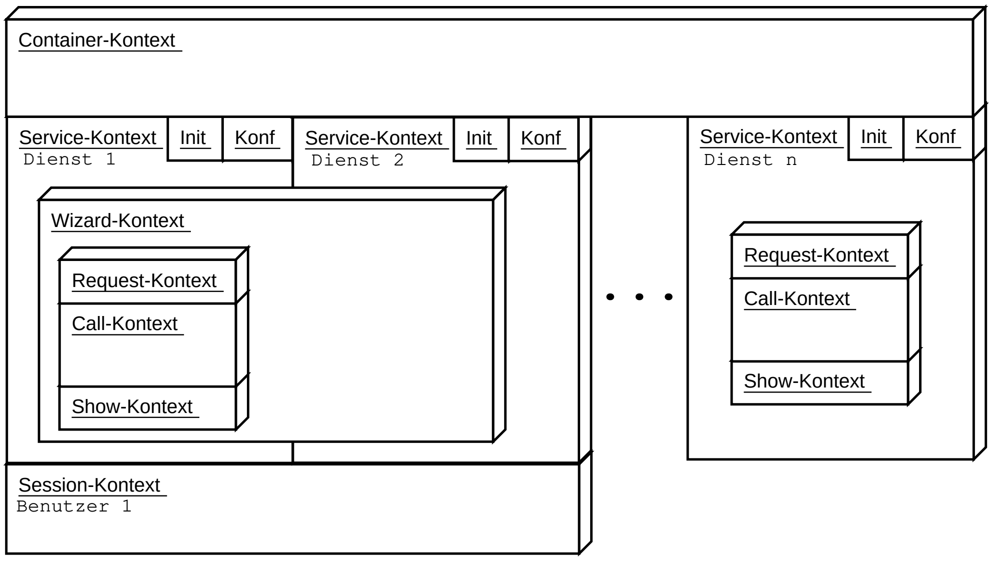


Abbildung 7.1: Die Datenkontexte – Übersicht

Sie dienen zur Speicherung, Zwischenspeicherung und Modifikation von Daten und Geschäftsobjekten, um die Applikationslogik, d.h. die Workflows der Applikation, zu spezifizieren bzw. zu realisieren.

Diese Datenkontexte unterscheiden sich in Erreichbarkeit und Verwendungszweck sowie der Lebensdauer der in ihnen enhaltenen Objekte. In den folgenden Abschnitten werden diese Kontexte und ihre wichtigsten Eigenschaften kurz beschrieben.

Der Konfigurations-Kontext: Eine web-basierte Applikation hat eine Konfigurations-Datei im XML-Format (hier: `web.xml`), in der die zur Applikation gehörigen Dienste und deren Konfigurationsdaten definiert werden – wie in Abb. 7.2 dargestellt.

```
<web-app>
...
  <servlet>
       <servlet-name>showNavbarPublic</servlet-name>
       <servlet-class>de.metaframe.ewis.WebInfoService</servlet-class>
       <init-param>
           <param-name>ServiceName</param-name>
           <param-value>showNavbarPublic</param-value>
       </init-param>
       <init-param>
           <param-name>Package</param-name>
           <param-value>de.unido.ls5.wis.ewis.application.templus</param-value>
       </init-param>
   </servlet>
...
</web-app>
```

Abbildung 7.2: Konfiguration eines Dienstes

Die Konfigurations-Kontexte der einzelnen Dienste werden von der XML-Umgebung `<web-app> ... </web-app>` eingefasst.

Es können zur Laufzeit ausschließlich Dienste ausgeführt werden, die in dieser Datei definiert sind. In der Konfiguration-Datei ist jedem Dienst ein Abschnitt gewidmet, den wir als *Konfigurations-Kontext* bezeichnen und der durch die XML-Umgebung `<servlet> ... </servlet>` eingeschlossen wird.

Jeder Dienst kann nur auf seinen eigenen *Konfigurations-Kontext*, d.h. die ihm zugeordneten Konfigurationsdaten zugreifen. Konfigurationsdaten werden mithilfe der XML-Umgebung `<init-param> ... </init-param>` angegeben, in der jeweils der Name und der Wert der Konfigurationsdaten definiert werden müssen.

Abb. 7.2 zeigt den Konfigurations-Kontext des Dienstes `showNavbarPublic` aus dem TEMPLUS Projekt, der das Anzeigen der öffentlichen Navigationsleiste realisiert und neben der minimalen Menge an Konfigurationsdaten für einen Dienst (bestehend aus dem Namen des Dienstes und dem Paket, in dem seine Implementierung abgelegt ist) keine weiteren Konfigurationsdaten besitzt.

Neben den Konfigurationsdaten kann für einen Dienst mithilfe der XML-Umgebung `<load-on-startup>...</load-on-startup>` festgelegt werden, wann er initialisiert werden soll. Diese wird innerhalb des Konfigurations-Kontextes des jeweiligen Dienstes plaziert. Der Wert `1` bedeutet dabei, dass der zugehörige Dienst initialisiert wird, sobald die Applikation auf dem Server gestartet wird, Wert `0` bzw. das Fehlen dieser XML-Umgebung bedeutet entsprechend, dass der Dienst initialisiert wird, wenn zum ersten Mal ein Zugriff auf ihn erfolgt.

Neben den einzelnen Diensten muss für die web-basierte Applikation noch die Zeitspanne festgelegt werden, wann eine nicht mehr benutzte Benutzer-Session ungültig wird und somit gelöscht werden kann. Der in Abb. 7.3 angegebene Ausschnitt aus der XML-Datei `web.xml` legt fest, dass jede unbenutzte Benutzer-Session nach 30 Minuten ungültig wird.

```
<session-config>
    <session-timeout>30</session-timeout>
</session-config>
```

Abbildung 7.3: Konfiguration der Benutzer-Sessions

Der Initialisierungs-Kontext gehört zu einem Dienst und wird innerhalb des zugehörigen *SLG*s als separater Pfad spezifiziert, der vom *Start*-Knoten des *SLG*s aus über den `init_service` Branch erreicht wird. Dieser Initialisierungspfad wird genau einmal ausgeführt. Je nach Konfiguration des Dienstes geschieht dies beim Starten der Applikation oder spätestens, wenn der Dienst zum ersten Mal angesprochen wird.

Während der Initialisierung eines Dienstes müssen alle initialen Daten, die für die durch den *SLG* des Dienstes realisierten Workflows vorausgesetzt werden, vorbereitet werden. Dazu gehören u.a. die Initialisierung des Datenbankzugriffs sowie der Datenbank-Administratoren, die den Datenbankzugriff koordinieren, d.h. die Abbildung zwischen den Geschäftsobjekten des Anwendungsbereiches - der Daten-Schicht - und der Persistenzschicht regeln.

Da der Initialisierungs-Kontext lediglich einmal ausgeführt wird und später im Kontrollfluss nicht mehr zugreifbar ist, müssen alle wichtigen dort vorbereiteten Daten (z.B. die Administratoren für die Geschäftsobjekte) in einen anderen, länger verfügbaren Datenkontext (z.B. den Container-Kontext) verschoben werden, bevor der Initialisierungs-Kontext wieder verlassen wird.

Der Container-Kontext ist aus allen Diensten einer Applikation heraus erreichbar, unabhängig davon, welcher Benutzer gerade mit dem jeweiligen Dienst interagiert, und hat dieselbe Lebensdauer wie die Applikation selbst. Dieser Datenkontext wird benutzt, um für alle Dienste relevante, globale Daten zu halten.

Daher ist dieser Datenkontext optimal für das Cachen von Daten geeignet, die zu einem beliebigen Zeitpunkt während der Laufzeit der Applikation von allen Diensten aus erreichbar sein müssen.

Um allen Diensten Zugriff auf die Datenbank zu gewähren sollten z.B. die Datenbank-Administatoren der Geschäftsobjekte im *Container-Kontext* plaziert werden, da sie die Schnittstelle zwischen Daten-Schicht und Persistenz-Schicht darstellen und damit erst die Ausführung von Datenbankanfragen und -modifikationen ermöglichen.

Auch allgemein verfügbare Daten, wie z.B. für die Visualisierung der personalisierten Navigation (siehe Abschnitt 11.4.3), die sich selten verändern, können in diesem Datenkontext zwischengespeichert werden, um die Effizienz der Applikation zu erhöhen.

In beiden Fällen wird der Container-Kontext als globaler Cache verwendet, wodurch der Speicherbedarf der Applikation gering gehalten wird bzw. die Anwortzeiten bei Verwendung gecachter Geschäftsobjekte verkürzt werden, da keine überflüssigen Datenbankzugriffe stattfinden.

Der Service-Kontext ist aus dem Dienst – d.h. *SLG* – heraus erreichbar, zu dem er gehört, unabhängig davon, welcher Benutzer gerade mit dem jeweiligen Dienst interagiert. Die Lebensdauer dieses Datenkontextes entspricht der Lebensdauer des zugehörigen Dienstes (also i.d.R. der Lebensdauer der Applikation). Dieser Datenkontext wird benutzt, um nur für den jeweiligen Dienst relevante, globale Daten zu halten.

Ein Kontakt-Dienst ermöglicht es z.B. per E-Mail Anfragen an einen bestimmten Ansprechpartner zu stellen. Im Rahmen dieses Dienstes kann die E-Mail-Adresse des jeweiligen Ansprechpartners im Service-Kontext gehalten werden, da sie ja nur innerhalb des Kontakt-Dienstes relevant und gültig ist.

Der Session-Kontext: Jedem Benutzer, der mit der web-basierten Applikation interagiert, wird eindeutig eine sogenannte Benutzer-*Session* zugeordnet. Diese ist verfügbar, solange der Benutzer mit der Applikation interagiert.

Die Zeitspanne, wie lange die Interaktion höchstens unterbrochen werden darf, ohne dass die *Session* ungültig bzw. gelöscht wird, wird ebenso wie die Konfigurationsdaten der einzelnen Dienste in der globalen Konfigurationsdatei der Applikation festgelegt.

Jeder Session ist ein Datenkontext, der sog. *Session-Kontext* zugeordnet. Zugriff auf den *Session-Kontext* eines Benutzers haben alle Dienste, mit denen dieser Benutzer interagiert. Der Session-Kontext dient somit als benutzer-spezifischer Cache.

Der *Session-Kontext* sollte verwendet werden, um benutzer-spezifische Daten zwischenzuspeichern, die häufig oder von allen Diensten, mit denen ein Benutzer interagiert, benötigt werden.

Beispiel 7.1:

Im Rahmen der TEMPLUS Applikation sind dies z.B. Personalisierungsdaten, konkret das zum aktuellen Benutzer gehörige `User`-Objekt (d.h. die Benutzerdaten) sowie die Rollen eines Benutzers. ⌟

Der Call-Kontext stellt den Standard-Kontext dar, in dem Daten während der Bearbeitung einer Benutzeranfrage gehalten werden sollten.

Dieser Datenkontext ist einem konkreten Dienst der Applikation und einem konkreten Benutzer zugeordnet, der mit diesem Dienst interagiert.

Seine Lebensdauer ist beschränkt auf den Bereich zwischen zwei direkt aufeinander folgenden Interaktionspunkten des zugehörigen Dienstes. Interaktionspunkte sind z.B. HTML-Seiten, die angezeigt werden, automatische Weiterleitungen oder Downloads.

Der Wizard-Kontext ist ein spezialisierter Datenkontext, der einem bestimmten Abschnitt der Applikationslogik, d.h. einem Teil-Workflow der Applikation zugeordnet ist.

Dieser Datenkontext ist nur für den ihm zugeordneten Benutzer und aus dem ihm zugeordenten Teil-Workflow heraus erreichbar und dient zur Zwischenspeicherung von Daten, die über mehrere HTML-Seiten bzw. Dienste hinweg benötigt werden, d.h. er ermöglicht das Aufsammeln von Daten während der zugehörige Benutzer mit den Diensten einer Applikation interagiert.

Beispiel 7.2:

Im Rahmen der TEMPLUS Applikation wird der Wizard-Kontext z.B. benutzt, um durch mehrere aufeinander folgende Auswahlprozesse (Auswahl einer Lehrveranstaltung, Auswahl einer Gruppe, Auswahl eines Benutzers) die Menge der angezeigten Daten entsprechend zu filtern. ⌟

Jeder Benutzer kann im Rahmen einer Applikation höchstens einen *Wizard-Kontext* besitzen. Die Lebensdauer des *Wizard-Kontexts* ist gekoppelt an die Lebensdauer der *Session* eines Benutzers. Beendet wird der Wizard-Kontext, sobald er nicht über einen Interaktionspunkt weitergereicht oder explizit mithilfe eines `EndWizard` *SIB*s beendet wird.

Der Request-Kontext bildet die Schnittstelle zwischen Benutzer und Applikationslogik, d.h. er ermöglicht den Datentransport von der Präsentations-Schicht in die Koordinations-Schicht einer Applikation.

So wird es möglich, Daten aus dem Formular einer Webseite oder über einen Link zu laden und dann innerhalb der Applikationslogik weiterzuverabeiten.

Die Lebensdauer dieses Datenkontextes entspricht der des Call-Kontextes.

Der Show-Kontext bildet die Schnittstelle zwischen Applikationslogik und Benutzer, d.h. er ermöglicht den Datentransport von der Koordinations-Schicht in die Präsentations-Schicht.

So wird es möglich, Daten aus der Applikationslogik innnerhalb einer HTML-Seite bzw. eines Downloads verfügbar zu machen und sie so dem Benutzer anzuzeigen bzw. sie zu speichern.

Applikations-spezifische Caches: Bei der Realisierung einer web-basierten Applikation kann die Verwendung von Caches aus Effizienzgründen notwendig sein. Cachen von Daten verringert den Speicherbedarf der Applikation und reduziert die Anzahl der notwendigen Datenbankzugrife.

Abhängig von dem Verwendungszweck eines Caches muss dieser in dem adäquaten Datenkontext plaziert werden.

- Ein *globaler Cache* enthält Daten, die von allen Benutzern aus allen Diensten heraus zugreifbar sein sollen. Daher muss diese Art von Cache im *Container-Kontext* plaziert werden.

  Beispiel 7.3:

  Im Rahmen der TEMPLUS Applikation sind dies z.B. die Datenbank--Administratoren der Geschäftsobjekte des Anwendungsbereiches sowie Daten zum Anzeigen der personalisierten Navigationsleisten. ⌟

- Ein *dienst-spezifischer Cache* enthält Daten, die spezifisch für einen konkreten Dienst sind, aus diesem heraus aber von allen Benutzern zugreifbar sein sollen. Daher muss ein derartiger Cache im *Service-Kontext* des jeweiligen Dienstes plaziert werden.

  Beispiel 7.4:

  Im Rahmen der TEMPLUS Applikation wird z.B. im Kontakt-Dienst die E-Mail-Adresse des Ansprechpartners im *Service-Kontext* dieses Dienstes gehalten. ⌟

- Je nach Applikation können *benutzer-spezifische Caches* nötig sein für das Sammeln und Zwischenspeichern häufig verwendeter benutzer-spezifischer Daten, z.B. persönliche Benutzer-Daten, eigene Rollen und Rechte.
  - Sollen diese Daten von allen Diensten aus zugreifbar sein, muss der zugehörige Cache im *Session-Kontext* plaziert werden.

    Beispiel 7.5:

    Im Rahmen der TEMPLUS Applikation sind dies z.B. das `User`-Objekt des aktuellen Benutzers sowie seine Rollen. ⌟

  - Sind diese Daten nur für einen Teil-Workflow relevant, sollten sie im *Wizard-Kontext* zwischengespeichert werden.

    Beispiel 7.6:

    Im Rahmen der TEMPLUS Applikation sind dies z.B. die ausgewählte Lehrveranstaltung, die ausgewählte Gruppe, ein ausgewählter Benutzer oder eine Rolle, die im weiteren Verlauf der Applikationslogik benötigt werden, um Daten zu filtern, oder aus einer Menge alternativer Wege im Kontrollfluss denjenigen zu bestimmen, der ausgeführt wird. ⌟

Kapitel 8

Software-Entwicklungsprozess mit *ABC-SD*

Für die Entwicklung web-basierter Applikationen wird in [11] ein arbeitsteiliger, rollen-basierter Software-Entwicklungsprozess eingeführt und detailliert beschrieben (siehe Abb. 8.1). Im Folgenden wird dieser Software-Entwicklungsprozess durch Verwendung der Abkürzung $PM_{ABC\text{-}SD}$ referenziert, da er ein Prozessmodell (kurz: *PM*) für die Entwicklung einer web-basierten Applikation mithilfe des *ABC-SD* definiert.

Die verschiedenen Rollen illustrieren dabei die unterschiedlichen Bereiche, aus denen während der Entwicklung web-basierter Applikationen mit *ABC-SD* Aufgaben erledigt werden müssen.

- Der **System-Ingenieur** analysiert das Problem, das mithilfe der web-basierten Applikation gelöst werden soll. Er hält den Kontakt mit dem Kunden und zeichnet verantwortlich für das externe Verhalten der zu entwicklenden Applikation, für die Umgebung, in welche die Applikation integriert werden soll, sowie für die Erfüllung von Performanz- und Sicherheits-Anforderungen.

- Der **OO-Spezialist** entwirft, implementiert und testet die Business Objekte des Anwendungsbereiches.

- Der ***SIB*-Integrator** realisiert die für den Anwendungsbereich benötigten, atomaren, applikations-spezifischen Komponenten (*SIB*s) auf Basis der verwendeten Business Objekte.

- Der **Anwendungsexperte** legt die von einer Applikation zur Verfügung gestellten Workflows fest und spezifiziert dann das Applikationsverhalten als *SLG* mit *ABC-SD*.

- Der **HTML-Designer** entwickelt die HTML-Seiten, welche die graphische Benutzeroberfläche der web-basierten Applikation darstellen.

In [12] wird eine Anwendung von $PM_{ABC\text{-}SD}$ im Rahmen der Entwicklung einer web-basierten Applikation mit *ABC-SD* beschrieben und detailliert illustriert, welche Schritte im Einzelnen dabei notwendig sind.

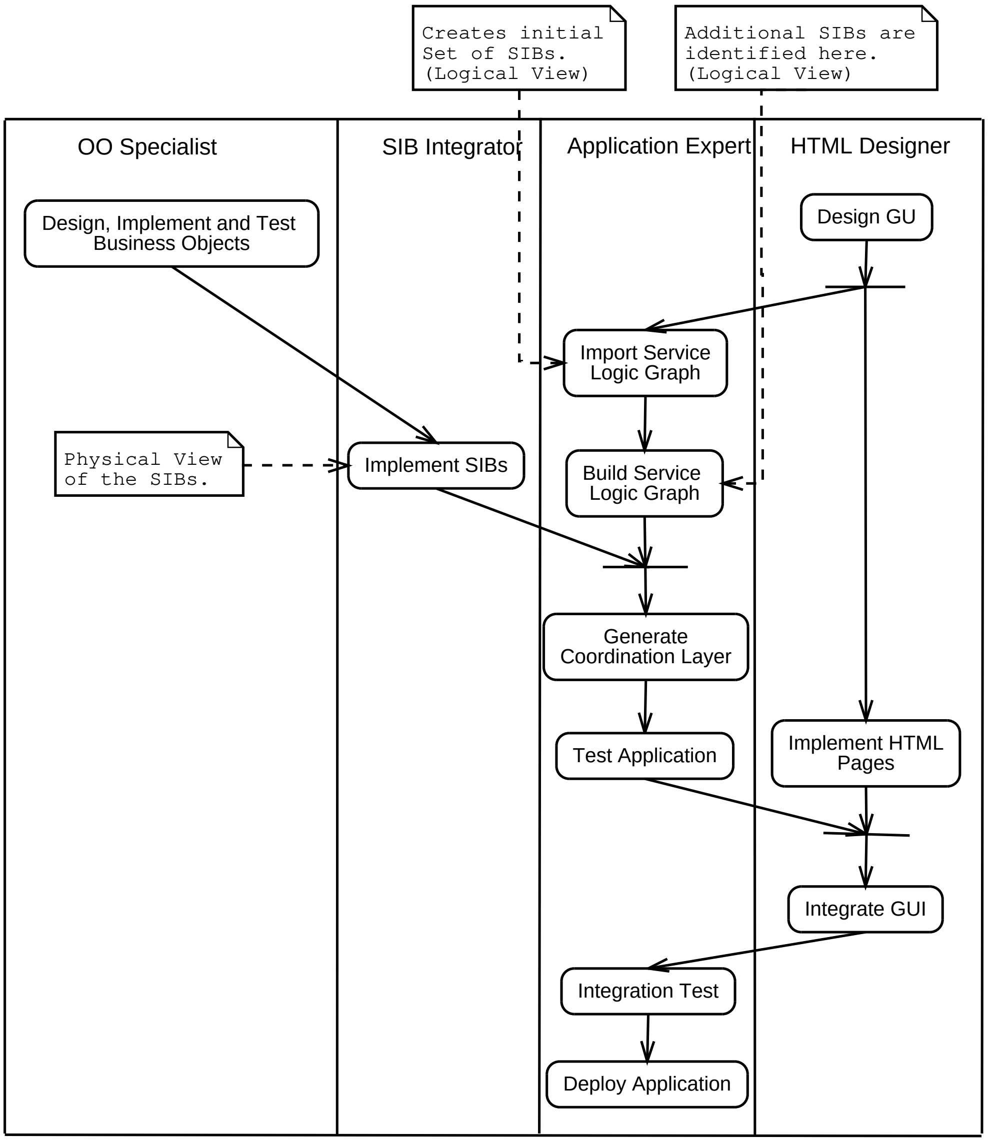


Abbildung 8.1: Der Software-Entwicklungsprozess $PM_{ABC\text{-}SD}$ [11]

Teil III

Framework – Entwicklung

Kapitel 9

Wiederverwendung von Diensten

Bei der Entwicklung einer web-basierten Applikation mit *ABC-SD* werden vordefinierte (d.h. applikations-unabhängige) oder selbst entwickelte (d.h. applikationsspezifische) Bibliotheken wiederverwendbarer Komponenten eingesetzt. Wie in Kapitel 5 beschrieben, können diese Komponenten *SIB*s oder Makros sein:

- ***SIB*s** stellen die atomaren funktionalen Einheiten für einen bestimmten Anwendungsbereich dar.
- Mithilfe von **Makros** lassen sich unter Verwendung von *SIB*s bzw. anderer Makros komplexe funktionale Einheiten für einen Anwendungsbereich spezifizieren.

Eine web-basierte Applikation wurde vor der Einführung des in dieser Arbeit vorgestellten Frameworks (siehe Abschnitt 2.1) mit *ABC-SD* immer als einzelner Dienst in Form eines ***SLG***s realisiert, welcher das Ablaufverhalten des zugehörigen Dienstes modelliert. Dabei werden *SIB*s und Makros eingesetzt, um den entsprechenden Graphen aufzubauen.

Die folgenden Abschnitte beschreiben die neuen Möglichkeiten der Wiederverwendung, welche sich durch die modulare Entwicklung web-basierter Applikationen mit *ABC-SD* als Dienstfamilie ergeben.

9.1 Ebenen der Wiederverwendbarkeit

Vor der Einführung des Frameworks für personalisierte, web-basierte Applikationen (siehe Abschnitt 2.1) wurden folgende drei Ebenen der Wiederverwendbarkeit von Komponenten unterschieden.

- Die ***Basiskomponenten*** (siehe Abschnitt 5.4) stellen die grundlegenden Funktionalitäten für web-basierte Applikationen in Form von *SIB*s zur Verfügung.

- ***Komponenten für verschiedene Aufgabenbereiche*** (*SIB*s und Makros) im Rahmen der Anwendungsbereiche web-basierter Applikationen erweitern die Menge der verfügbaren funktionalen Einheiten – wie in Abschnitt 5.4 beschrieben.

- ***Applikations-spezifische Komponenten*** (*SIB*s und Makros) können für eine neue Applikation definiert werden, um benötigte Funktionalitäten zu realisieren, die nicht bzw. nicht adäquat durch Verwendung oder Kombination von Komponenten aus einer der beiden anderen Ebenen umgesetzt werden können.

Alle Komponenten dieser drei Ebenen können verwendet werden, um das Verhalten einer neuen Applikation in Form eines einzelnen *SLG*s zu modellieren. Abb. 9.1 illustriert die beschriebenen Ebenen der Wiederverwendbarkeit am Beispiel der vordefinierten Bibliotheken wiederverwendbarer Komponenten aus dem *ABC-SD* (siehe Abschnitt 5.4) und setzt diese in Bezug zu einer neuen Applikation.

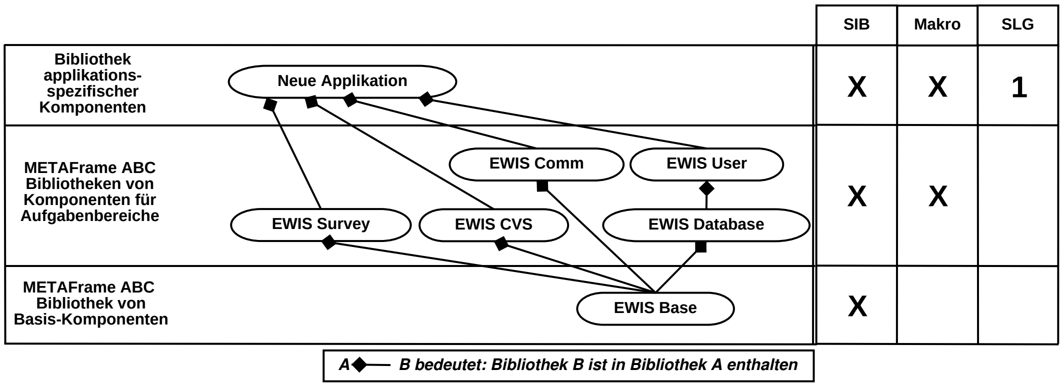


Abbildung 9.1: Bibliotheken und Komponenten vor Einführung des Frameworks

Mit der Entwicklung des Frameworks für personalisierte, web-basierte Applikationen vollziehen wir den Übergang von einem Dienst zu einer Dienstfamilie bei der Modellierung der Applikationslogik, d.h. des Ablaufverhaltens einer Applikation – wie bereits in Abschnitt 2.1 dargestellt.

Ein Makro muss – bei Verwendung während der Modellierung der Applikationslogik – stets in einen *SLG* eingebaut werden. Dabei kann eine Konfiguration des Makros durch das Setzen von Werten für evtl. vorhandene Parameter nötig sein. Außerdem muss die korrekte Funktionsweise eines Makros bzgl. der Interaktion mit den übrigen innerhalb des *SLG*s verwendeten Komponenten überprüft und garantiert werden. Dies geschieht durch Validierung zur Designzeit bzw. beim Testen der fertigen Applikation.

Dienste dagegen sind eigenständig ausführbar. Einmal validiert und getestet, können sie als zuverlässig arbeitende Komponenten beim modularen Aufbau einer personalisierten, web-basierten Applikation aus einer Menge von Diensten wiederverwendet werden. So wird eine Wiederverwendung auf höherem Niveau als bisher ermöglicht. Neben *SIB*s und Makros stehen nun auch Dienste als wiederverwendbare Komponenten zur Verfügung. Hierdurch kann der Aufwand für die Entwicklung, die Validierung und das Testen der gesamten Applikation reduziert werden.

Die Menge der Ebenen, welche die Wiederverwendbarkeit im Rahmen personalsierter, web-basierter Applikationen beschreiben, wird nun erweitert. Diese Erweiterung beinhaltet eine Bibliothek von Komponenten aus dem Framework für personalisierte, web-basierte Applikationen, welche neben *SIB*s und Makros auch erstmals Dienste als (wiederverwendbare) Komponenten bereitstellt.

Abb. 9.2 illustriert die beschriebenen vier Ebenen der Wiederverwendbarkeit am Beispiel der Bibliotheken von Komponenten, welche im Rahmen des WIS-Teilprojekts Nr. 6 eingesetzt werden.

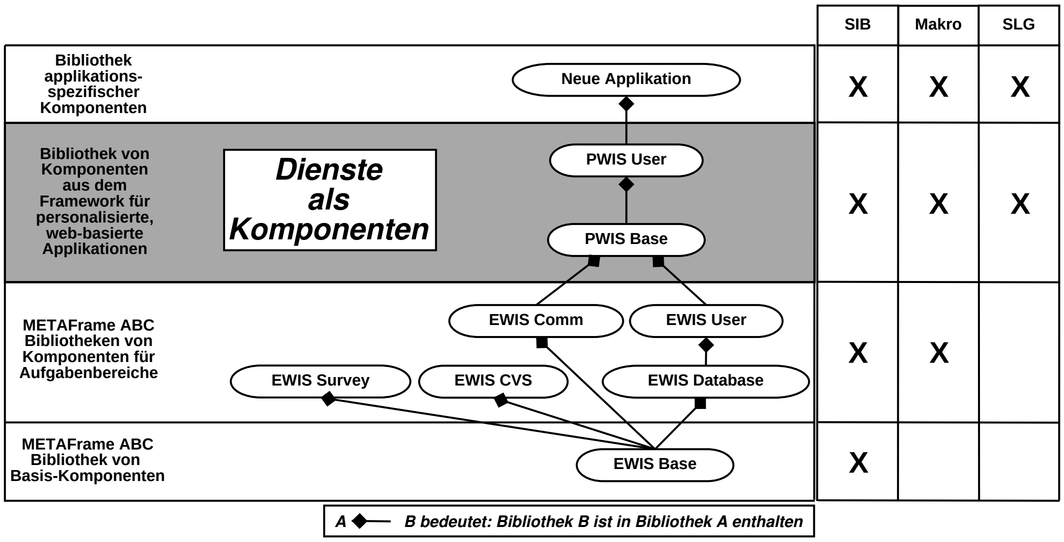


Abbildung 9.2: Bibliotheken und Komponenten nach Einführung des Frameworks

In dem folgenden Abschnitt 9.2 werden einige Szenarien im Rahmen web-basierter Applikationen vorgestellt, die sich für eine Wiederverwendung von Diensten als Komponenten eignen.

9.2 Szenarien

Für die Wiederverwendung von Diensten als Komponenten eignen sich grundsätzlich die in den folgenden Abschnitten beschriebenen Szenarien. Hierfür unterteilen wir die Dienste, welche Bestandteil der Realisierung einer personalisierten, web-basierten Applikation sind, anhand ihres Verwendungsszweckes in zwei Gruppen – externe Dienste und interne Utility-Dienste.

9.2.1 Externe Dienste

Externe wiederverwendbare Dienste realisieren Features eines Anwendungsbereiches, die unverändert von einer personalisierten, web-basierten Applikation in eine andere übernommen werden können. Die Ausführung dieser Art von Diensten wird explizit von einem Benutzer angestoßen, d.h. es handelt sich hierbei um Dienste, die den Benutzern über eine öffentliche oder personalisierte Navigationsleiste angeboten werden.

Beispiel 9.1:

Die 'Basisdienste' des in Abschnitt 2.1 beschriebenen Frameworks für personalisierte, web-basierte Applikationen – welche Administrations-, Verwaltungs-, und Navigationsfunktionalität realisieren – stellen wiederverwendbare externe Dienste dar. Sie können im Rahmen der Entwicklung personalisierter, web-basierter Applikationen mit *ABC-SD* für die Realisierung beliebiger web-basierter Applikationen genutzt werden. Abb. 9.3 illustriert,

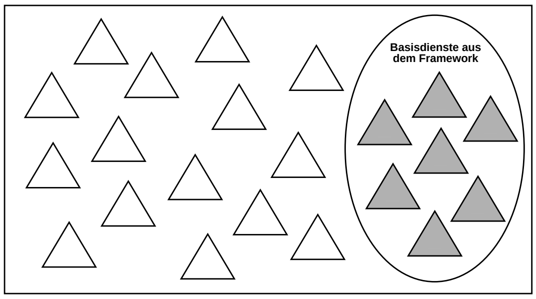


Abbildung 9.3: 'Basisdienste' als Teil der Gesamtfunktionalität einer Applikation

am Beispiel des Frameworks, dass die Menge der wiederverwendeten Dienste einen Teil der Gesamtfunktionalität der zu entwickelnden personalisierten, web-basierten Applikation bildet. ⌟

9.2.2 Interne Utility-Dienste

Interne Utility-Dienste realisieren Teile von externen Diensten, die in sich geschlossene Funktionalitäten darstellen, häufig innerhalb des Applikationsverhaltens genutzt werden und eine klar definierte Schnittstelle besitzen. Die Ausführung dieser Utility-Dienste wird implizit während der Ausführung eines externen Dienstes angestoßen, d.h. es handelt sich hierbei um Dienste, die den Benutzern nicht über eine Navigationsleiste angeboten werden. Dabei ist die Verwendung desselben Utility-Dienstes in verschiedenen Features möglich. Informationen in Form von Daten können an einen Utility-Dienst übertragen werden, vergleichbar einem Funktionsaufruf mit Parametern in Programmiersprachen.

Beispiel 9.2:

Bei der Unterstützung der mittelbaren Kommunikation zwischen den Personen, die an den Geschäftsprozessen im Rahmen der Lehre an einer Universität beteiligt sind, durch elektronische Medien spielt das Versenden von E-Mails eine wesentliche Rolle.

Da wir web-basierte Applikationen entwickeln, beschränken wir uns in diesem Beispiel auf das Versenden von E-Mails innerhalb eines Features des Anwendungsbereiches.

Bei der Entwicklung eines E-Mail Utility-Dienstes spielen die typischen Eigenschaften einer E-Mail eine wesentliche Rolle. Eine E-Mail setzt sich zumindest zusammen aus Text, Absender und Empfänger. Zudem kann der Absender entscheiden, ob er eine Kopie der E-Mail erhalten möchte. Wird ein E-Mail-Dienst innerhalb einer personalisierten, web-basierten Applikation angeboten, spielt außerdem seine Sichtbarkeit eine Rolle, d.h. ob der E-Mail Utility-Dienst in einem öffentlichen oder in einem personalisierten externen Dienst verwendet wird.

- *Sichtbarkeit*: Die betrachtete E-Mail-Utility kann Bestandteil eines *öffentlichen* oder eines *personalisierten* externen Dienstes sein.

- *Absender*: Die E-Mail-Adresse des Absenders kann *benutzerdefiniert* sein, d.h. deren Eingabe erfolgt als Freitext, oder wird durch die Applikation fest vorgegeben, z.B. die E-Mail-Adresse des aktuell eingeloggten Benutzers.

- *Empfänger*: Die Menge der Empfänger kann *ein- oder mehrelementig* sein. Außerdem unterscheiden wir die beiden Fälle, dass die Menge der Empfänger entweder im Rahmen der Applikation *fest* vorgegeben oder innerhalb einer vorgegebenen größeren Menge *wählbar* ist.

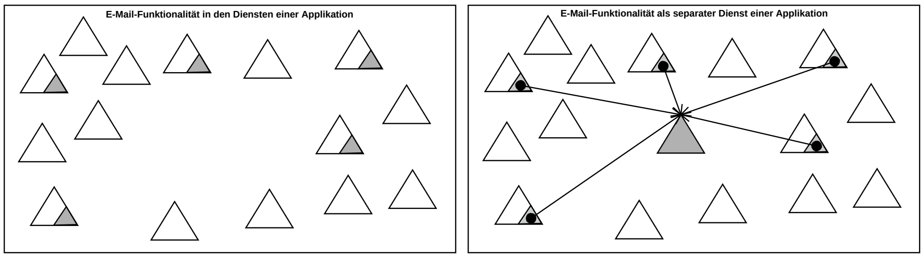
- *E-Mail-Text*: Der E-Mail-Text kann *benutzerdefiniert* sein, wenn seine Eingabe über ein Online-Formular erfolgt, oder – bei System- bzw. Statusmeldungen – durch die Applikation *fest* vorgegeben sein.

- *Kopie*: Generell wird keine Kopie der jeweiligen E-Mail an den Absender geschickt.

Manche E-Mail-Funktionalitäten überlassen aber dem Benutzer die Entscheidung, ob er eine Kopie erhalten möchte.

Wird ein Typ von E-Mail-Funktionalität innerhalb sehr vieler Dienste einer Applikation verwendet, so sind diese Teile der zur Applikation gehörigen Dienste durch entsprechend identische Teilgraphen realisiert – wie in Abb. 9.4 (a) dargestellt. Dadurch steigt die Größe der *SLG*s einer Applikation unnötig. Auch bei Verwendung von Makros, muss für jeden dieser *SLG*s dieselbe E-Mail-Funktionalität validiert und getestet werden.

Um diese Art von Redundanz auf Graphebene und bei der arbeitsteiligen Entwicklung der Applikation zu vermeiden kann eine E-Mail-Funktionalität ebenso gut als separater Dienst der Applikation realisiert werden, der dann nach entsprechender Vorbereitung der benötigten Daten aus allen Diensten heraus angesprochen werden kann. Abb. 9.4 (b) skizziert diese Alternative der Umsetzung von E-Mail-Funktionalität innerhalb einer Applikation.



(a) innerhalb von Diensten (b) als separater Dienst

Abbildung 9.4: E-Mail-Funktionalität einer Applikation

Abb. 9.5 klassifiziert einige Beispiele für Features mit integrierter E-Mail-Funktionaliät aus dem TeMPlUs Projekt unter Verwendung der hier angegebenen typischen Eigenschaften von E-Mails.

⌟

externer Dienst: Feature (E-Mail)	**Sichbarkeit**	**Absender**	**Empfänger**		**Text**	**Kopie**
contact: 'Kontakt' (über Online-Formular)	öffentlich	benuter-definiert	fest	einelementig	benutzer-definiert	nein
register: 'Registrierung' (Bestätigung und Registrierungsdaten an den Benutzer)	öffentlich	fest	fest	einelementig	fest	nein
courseSendMail: 'E-Mail an die Teilnehmer einer Lehrveranstaltung senden' (über Online-Formular)	personalisiert	fest	wählbar	mehrelementig	benutzer-definiert	wählbar
groupSendMail: 'E-Mail an die Teilnehmer einer Gruppe einer Lehrveranstaltung senden' (über Online-Formular)	personalisiert	fest	wählbar	mehrelementig	benutzer-definiert	wählbar
authenticateUser: 'Benutzer authentifizieren' (Statusmeldung an die betroffenen Benutzer)	personalisiert	fest	wählbar	einelementig	fest	nein
lockUser: 'Benutzer sperren' (Statusmeldung an die betroffenen Benutzer)	personalisiert	fest	wählbar	einelementig	fest	nein
distribution: 'Gruppeneinteilung initiieren' (vier verschiedene Statusmeldungen zum Fortschritt bei der Verteilung)	personalisiert	fest	fest	einelementig	fest	nein
viewCourse: 'Daten einer Lehrveranstaltung anzeigen' (Lehrpersonal kontaktieren über Online-Formular)	personalisiert	fest	wählbar	einelementig	benutzer-definiert	wählbar

Abbildung 9.5: E-Mail-Funktionalität in TEMPLUS Diensten

Kapitel 10

Klassifikation von Komponenten

Taxonomien stellen eine hierarchische Einteilung bzw. Klassifikation von Dingen in Gruppen (*griech.: Taxon*) bzw. Mengen dar.

In einer Taxonomie unterscheidet man zwischen den *Mengen*, die Einordnungskriterien beschreiben und hierarchisch angeordnet werden, und den *Instanzen*, welche die Dinge repräsentieren, die mithilfe der Taxonomie klassifiziert werden.

Eine Taxonomie repräsentiert dabei einen DAG[1]. Dies kann man sich wie ein virtuelles Filesystem vorstellen. Dort können konkrete Dateien ebenfalls in beliebig viele Verzeichnisse eingeordnet werden, und auch das Einfügen von Unterverzeichnissen ist möglich.

Im Rahmen des *ABC* kommen Taxonomien bereits an vielen Stellen zum Einsatz ([51, 8, 76, 2]).

Mithilfe einer Taxonomie, welche einer web-basierten Applikation zugeordnet wird, können demnach auch die für eine web-basierte Applikation wesentlichen Aspekte festgelegt sowie die zur Verfügung stehenden und verwendeten Komponenten anhand dieser Aspekte klassifiziert werden.

Für jede personalisierte, web-basierte und als Dienstfamilie konzipierte Applikation wird nun eine zugehörige Taxonomie definiert, in welche die verwendeten Komponenten – d.h. *SIB*s, Makros und Dienste – entsprechend ihrer Eigenschaften und Beziehungen untereinander eingeordnet werden. Diese Klassifizierung dient einerseits der Dokumentation, andererseits können die dadurch festgelegten Zusammenhänge auch im Rahmen der Entwicklung und Validierung einer Applikation gewinnbringend eingesetzt werden.

Die Abschnitte 10.1 und 10.2 gehen näher auf die Möglichkeiten bei der Klassifikation der Komponenten einer personalisierten, web-basierten Applikation ein.

[1] Directed Acyclic Graph.

10.1 Klassifikation von *SIB*s und Makros

Im *ABC-SD* wird eine Applikation innerhalb eines *ABC*-Projektes realisiert, welches eine Bibliothek applikations-spezifischer Komponenten (d.h. *SIB*s und Makros) beinhaltet – wie in Abschnitt 5.1 beschrieben.

Die verfügbaren *SIB*s und Makros werden dem Anwendungsexperten über eine *SIB*-Palette bereitgestellt (siehe Abschnitt 5.1 und Abb. 5.2).

Die Präsentation über die *SIB*-Palette basiert auf einer einfachen Klassifizierung, wobei jede dieser Komponenten genau einer Komponentenklasse zugeordnet ist. Diese Zuordnung wird bei der Spezifikation eines *SIB*s bzw. Makros in der Datei `<component_name>.sib` durch eine Zeile der Form `CLS <class_name>` ausgedrückt. Eine umfassende Beschreibung zu der Spezifikation von *SIB*s wird in [55] gegeben.

Abb. 10.1 und Abb. 10.2 zeigen jeweils ein Beispiel für eine Komponentenspezifikation eines *SIB*s bzw. eines Makros.

Beispiel 10.1:

Der *SIB* `ServiceProperty2InitContext` (siehe Abb. 10.1) ist eine im Projekt PwisBase definierte Komponente, die Konfigurationsdaten (`keys[]m_property_keys`) im Intitialisierungs-Kontext als Daten (`keys[]m_init_context_keys`) verfügbar macht.

```
SIB  ServiceProperty2InitContext
CLS  Context
PAR  keys DIM * 1
PAR  keys[]m_init_context_keys STR 100 ""
PAR  keys[]m_property_keys STR 100 ""
BR   dflt
PROP "java_package" "de.unido.ls5.ewis.application.pwisbase.sib"
```

Abbildung 10.1: Beispiel – *SIB*-Spezifikation

Da dieser *SIB* Daten von einem Datenkontext (dem Konfigurations-Kontext) in einen anderen Datenkontext (den Intitialisierungs-Kontext) kopiert, also auf verschiedenen Datenkontexten arbeitet, ist er in die Komponentenklasse (`CLS`) `Context` eingeordnet – wie in der zweiten Zeile der *SIB*-Spezifikation dargestellt.

⌟

Beispiel 10.2:

Das Makro `InitUserCache` (siehe Abb. 10.2) ist im PwisBase Projekt definiert und dient der Initialisierung des benutzer-spezifischen Caches. Dieser ist im Session-Kontext eines jeden Benutzers lokalisiert, wird beim Einloggen initialisiert und enthält häufig benötigte persönliche Daten des Benutzers (wie Name, Vorname, zugewiesene Rollen).

```
SIB  InitUserCache
CLS  Cache
MAC  "InitUserCache.smg"
BR   ok
```

Abbildung 10.2: Beispiel – Makro-Spezifikation

Da dieses Makro auf einen Cache zugreift und diesen modifiziert, ist es in die Komponentenklasse (`CLS`) `Cache` eingeordnet – wie in der zweiten Zeile der Makro-Spezifikation dargestellt.

⌟

Die Praxis zeigt jedoch, dass diese strikte und damit sehr eingeschränkte Zuordnung einer Komponente zu genau einer Komponentenklasse einen entscheidenen Nachteil in sich birgt. Wichtige Informationen über die Komponente bleiben unberücksichtigt bzw. werden wegabstrahiert, da der Anwendungsexperte gezwungen wird, sich für eine einzige Komponentenklasse zu entscheiden, wenn er eine neue Komponente innerhalb einer applikations-spezifischen Bibliothek im Rahmen des *ABC-SD* spezifiziert.

*SIB*s und Makros können aber viele Eigenschaften besitzen, die als Kriterien für eine Klassifizierung oder Suche benutzbar sein sollten, je nach dem, aus welchem Blickwinkel man eine solche Komponente betrachtet.

Einige dieser Eigenschaften, anhand derer *SIB*s und Makros klassifiziert werden, sind beispielsweise:

- die Projektzugehörigkeit der Komponente, d.h. in welchem *ABC-SD*-Projekt diese definiert ist,
- ob es sich bei der Komponente um einen *SIB* oder ein Makro handelt,
- die Parameter und Branches der Komponente,
- lesender bzw. schreibender Zugriff auf die verschiedenen Datenkontexte im Rahmen des *ABC-SD*,
- ob die Komponente auf Dateien zugreift, diese erzeugt oder modifiziert,
- lesender bzw. schreibender Zugriff auf eine Datenbank,

- ob die Komponente Benutzerinteraktion ermöglicht oder nur systeminterne Aktionen realisiert.
- auf welchen Typen von Geschäftsobjekten eine Komponente arbeitet und wie sie mit ihnen umgeht (z.B. lesender bzw. schreibender Zugriff, Erzeugung oder Löschen), sowie
- in welche Teilgebiete des jeweiligen Anwendungsbereichs der Applikation eine Komponente eingeordnet werden kann.

  Beispiel 10.3:

  Das TEMPLUS Projekt beschäftigt sich mit der Organisation und Durchführung von Lehrveranstaltungen und ihren Gruppen und bietet darüber hinaus ein flexibles Benutzer- und Rollen-Management, hier wäre also z.B. eine Klassifizierung von Komponenten nach den Teilgebieten *Lehrveranstaltung*, *Gruppe*, *Benutzer* und *Rolle* des Anwendungsbereichs 'Lehremanagement an einer Universität' der Applikation möglich. ⌟

- Falls es sich bei der Komponente um ein Makro handelt, ist sicherlich auch noch die Information wichtig, welche anderen Typen von Komponenten bei der Realisierung des Makros verwendet werden.

Die Liste der Eigenschaften für die Klassifikation von *SIB*s und Makros lässt sich – insbesondere mit applikations-spezifischen Einordnungskriterien – beliebig fortsetzen und erweitern. In der Regel besitzt eine Komponente stets mehrere Kriterien, anhand derer sie klassifiziert wird. Die folgenden Beispiele belegen dies anhand der zuvor betrachteten Komponenten:

Beispiel 10.4:

Für den *SIB* `ServiceProperty2InitContext` aus Abb. 10.1 stehen – neben der Zuordnung zu der Komponentenklasse `Context` – folgende zusätzlichen Informationen für eine weiterführende Klassifizierung zur Verfügung:

- Bei der Komponente handelt es sich um einen *SIB*.
- Die Komponente ist im Projekt PWISBase definiert.
- Konfigurationsdaten werden benötigt, d.h. es erfolgt ein lesender Zugriff auf den Konfigurations-Kontext.
- Es erfolgt ein schreibender Zugriff auf den Initialisierungs-Kontext.

- Die Komponente kopiert Daten von einem Datenkontext in einen anderen.
- Die Komponente erhält den Typ der kopierten Objekte.

┘

Beispiel 10.5:

Für das Makro `InitUserCache` aus Abb. 10.2 stehen – neben der Zuordnung zu der Komponentenklasse `Cache` – folgende zusätzlichen Informationen für eine weiterführende Klassifizierung zur Verfügung:

- Bei der Komponente handelt es sich um ein Makro.
- Die Komponente ist im Projekt PWISUser definiert.
- Es erfolgt ein lesender Zugriff auf die Datenbank.
- Es erfolgt ein schreibender Zugriff auf den Session-Kontext.
- Die Komponente arbeitet auf Geschäftsobjekten vom Typ `User` bzw. `UserGroup`.

┘

Makros und *SIB*s können nun unter Berücksichtigung all dieser Informationen in die dem zugehörigen Projekt zugrunde liegende Taxonomie (siehe Abschnitt 10.3) eingeordnet werden.

10.2 Klassifikation von Diensten

Bei der Entwicklung web-basierter Applikationen mit *ABC-SD* war vor der Einführung des Frameworks eine Klassifikation von Diensten nicht notwendig, da eine Applikation stets als einzelner Dienst realisiert wurde.

Nach der Einführung des Frameworks in Abschnitt 2.1 dieser Arbeit ist es nun auch möglich, web-basierte Applikationen mithilfe mehrerer Dienste zu realisieren.

Einige dieser Eigenschaften, anhand derer Dienste klassifiziert werden, sind im Folgenden zusammengestellt:

- Der bei der Klassifikation von Diensten wichtigste Aspekt ist sicherlich die Zugreifbarkeit. Grundsätzlich werden dabei verschiedene Arten von Diensten unterschieden:

  - Öffentliche Dienste,
  - Private Dienste,
  - Dienste, die eine Login-Aktion beinhalten und demnach öffentlich zugänglich sein müssen, und
  - Dienste, die eine Logout-Aktion realisieren und demnach im privaten Bereich einer Applikation zu finden sind.
  - Darüberhinaus besitzt jede web-basierte Applikation einen Mgmt-Dienst, der den initialen Dienst – d.h. den Einstiegspunkt in die gesamte Applikation – darstellt und demnach öffentlich zugänglich sein muss.

- Die Projektzugehörigkeit eines Dienstes gibt an, in welchem *ABC-SD*-Projekt dieser definiert ist.

- Bei der Realisierung eines Dienstes können andere Komponenten (d.h. *SIB*s und Makros) verwendet werden. Hieraus erhält der Dienst implizit die Eigenschaften, welche für die in ihm enthaltenen Komponenten spezifiziert worden sind (siehe Abschnitt 10.1).

- Vorbedingungen, die erfüllt sein müssen, damit dieser Dienst ausgeführt werden kann, können im Rahmen der Taxonomie dokumentiert werden.

- Eine Beschreibung der Nachbedingungen, die nach Ausführung des Dienstes gelten, kann ebenfalls im Rahmen der Taxonomie dokumentiert werden.

- Ein Dienst kann andere Dienste referenzieren, d.h. den Kontrollfluss an sie übergeben.

- Eine Einordnung des Dienstes in Teilgebiete des jeweiligen Anwendungsbereichs der Applikation führt zu einer Strukturierung der Gesamtfunktionalität einer Applikation.

Ebenso wie für Makros und *SIB*s erscheint daher auch für die Dienste einer personalisierten, web-basierten und als Dienstfamilie konzipierten Applikation eine Einordnung in die dem zugehörigen Projekt zugrunde liegende Taxonomie (siehe Abschnitt 10.3) sinnvoll.

Beispiel 10.6:

Für folgende Dienste aus der TEMPLUS Applikation werden einige Klassifizierungskriterien angegeben, die jeweils unterstrichen sind:

- Der Dienst *registerAsStudent* ist ein öffentlicher Dienst, mithilfe dessen Studierende sich bei der Applikation als Benutzer mit der Rolle Student registrieren können. Es handelt sich also um einen Dienst aus dem Teilgebiet 'Benutzer' des Anwendungsbereiches. Der Dienst erzeugt neue Geschäftsobjekte vom Typ `User` und `StudentData`, welche die Benutzer- und Studierendendaten aufnehmen. Es erfolgt ein schreibender Zugriff auf die Datenbank, um diese Informationen zu speichern.

- Der Dienst *showParticipantsCourse* ist ein privater Dienst und dient dem Organisator einer Lehrveranstaltung dazu, sich deren Teilnehmerliste anzeigen zu lassen. Es handelt sich hierbei demnach um einen Dienst aus dem Teilgebiet 'Lehrveranstaltung' des Anwendungsbereiches. Der Dienst verwendet Geschäftsobjekte vom Typ `Course` und `Participant`, welche Daten für eine Vorlesung bzw. einen Teilnehmer dieser beinhalten. Es erfolgt ein lesender Zugriff auf die Datenbank, um die Informationen anzuzeigen.

⌟

10.3 Taxonomie

Die Untersuchung der Klassifikationskriterien für *SIB*s und Makros (siehe Abschnitt 10.1) bzw. Dienste (siehe Abschnitt 10.2) zeigt, dass eine einfache Klassifizierung der Komponenten anhand eines einzigen Kriteriums – wie sie bei der Spezifikation von *SIB*s und Makros verwendet wird ([55]) – nicht ausreicht.

Wir benötigen also eine mehrstufige Taxonomie für die Klassifizierung der Komponenten, um eine hierarchische Einordnung dieser Komponenten anhand mehrerer Kriterien zu ermöglichen.

Bei der Klassifikation von Dingen mithilfe einer Taxonomie liegt der Fokus immer auf genau einer Art von Dingen, also in unserem Beispiel Komponenten. Die bestehenden *ABC-SD* Projekte werden dabei beispielsweise als Mengen dargestellt, um die Projektzugehörigkeit der Komponenten ausdrücken zu können.

Beispiel 10.7:

Wie in den Abschnitten 10.1 und 10.2 beschrieben, gibt es für *SIB*s, Makros und Dienste neben ihrem Komponententyp noch weitere Kriterien, anhand derer diese Komponenten klassifiziert werden können. Komponenten sind beispielsweise immer innerhalb einer Bibliothek definiert, welche durch ein *ABC-SD* Projekt zur Verfügung gestellt wird. Zwischen den Mengen 'Component' und 'Project' besteht also eine Beziehung, welche die Projektzugehörigkeit einer Komponente beschreibt und durch die Relation `defined_in` $\subseteq$ `Component` $\times$ `Project` ausgedrückt wird.

Die Verwaltung der Komponenten inklusive all ihrer Klassifizierungkriterien kann daher mithilfe einer Taxonomie erfolgen. Abb. 10.3 zeigt einen Ausschnitt aus einer Taxonomie für die Klassifizierung von Komponenten. Dieser enthält eine Unterscheidung nach

Komponententyp sowie die *ABC-SD*-Bibliotheken.

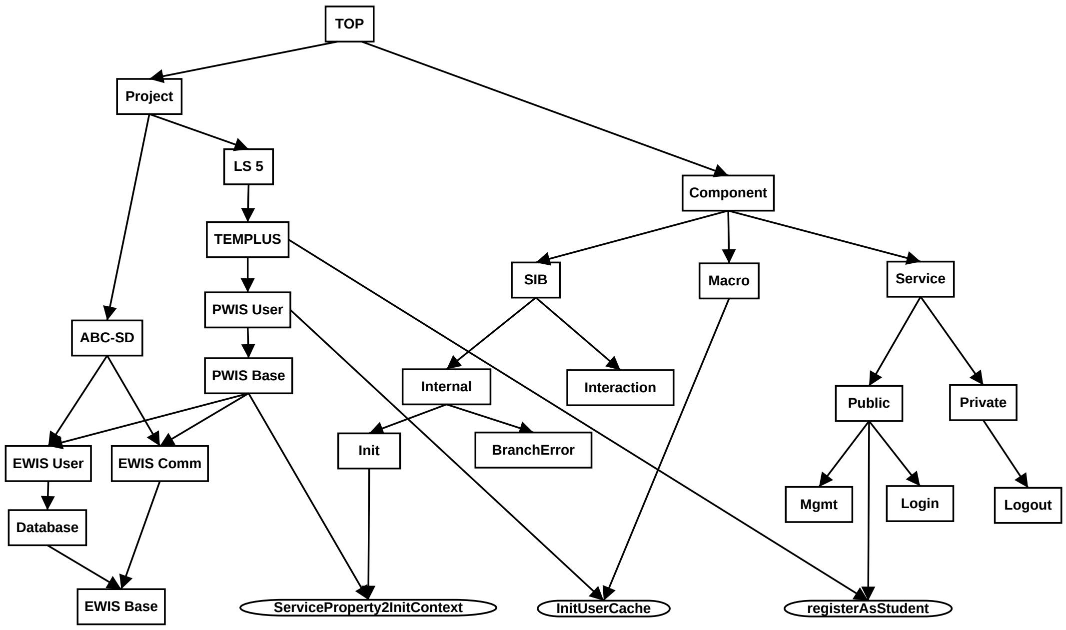


Abbildung 10.3: Ausschnitt aus einer Taxonomie für Komponenten

Die Komponente `ServiceProperty2InitContext` aus Bsp. 10.4 ist in die Mengen 'SIB' und 'PWIS Base' einzuordnen.

Die Komponente `InitUserCache` aus Bsp. 10.5 ist in die Mengen 'Macro' und 'PWIS User' einzuordnen.

Die Komponente *registerAsStudent* aus Bsp. 10.6 ist in die Mengen 'Public' und 'TEMPLUS' einzuordnen.

Natürlich müssen auch die anderen Klassifizierungskriterien dieser Komponenten bei einer Einordnung in eine Taxonomie berücksichtigt werden. Abb. 10.3 stellt aber lediglich einen Ausschnitt dar, daher wird auf die übrigen Klassifizierungskriterien hier nicht weiter eingegangen. Diese sind aber in den Abschnitten 10.1 bzw. 10.2 näher beschrieben. ⌟

Wir wollen die Taxonomie verwenden, um alle Dinge bzw. Daten für eine im Rahmen eines *ABC*-Projektes zu entwickelnde, personalisierte, web-basierte Applikation strukturiert darzustellen und zu verwalten.

Folgende verschiedene Arten von Daten sind im Rahmen des *ABC-SD* von Interesse:

- *ABC*-Projekte
- Datenkontexte

- Komponenten für die Entwicklung web-basierter Applikationen sowie deren Ein- und Ausgabeparameter, die sich auf Variablen beziehen, welche in den Datenkontexten einer Applikation gehalten werden

- Teilgebiete und Typen von Geschäftsobjekten aus dem Anwendungsbereich einer web-basierten Applikation

Neben diesen unterschiedlichen Typen von Daten sind auch die Beziehungen wichtig, welche zwischen Daten unterschiedlichen Typs bestehen.

Bestehende Beziehungen zwischen solchen Mengen werden dabei durch die Definition einer neuen Menge als Klassifizierungskriterium innerhalb der Taxonomie ausgedrückt.

Die allgemeinen Zusammenhänge, die beim Einsatz des *ABC-SD* für die Entwicklung web-basierter Applikationen wesentlich sind, werden folgendermaßen innerhalb der Taxonomie repräsentiert:

- *ABC* Projekte werden durch die Menge 'Project' repräsentiert.

- Die im Rahmen des *ABC-SD* existierenden Datenkontexte (siehe Kapitel 7) werden mithilfe der Menge 'DataContext' dargestellt.

- Komponenten, die bei der Realisierung einer web-basierten Applikation verwendet werden, sind unter der Menge 'Component' zusammengefasst.

- Die Ein- und Ausgabeparameter von Komponenten werden durch die Menge 'ContextKey' beschrieben.

- Parameter von Komponenten besitzen einen Typ, der innerhalb der Taxonomie mithilfe der Menge 'Type' dargestellt wird.

- Der Anwendungsbereich einer Applikation wird durch die Menge 'Domain' beschrieben und kann durch Einfügen von Untermengen in Teilgebiete unterteilt werden.

Zwischen diesen innerhalb der Taxonomie abgebildeten Objekten bestehen eine Reihe von Beziehungen, die als Relationen definiert werden und innerhalb der Taxonomie durch das Einfügen entsprechender Teilmengen abgebildet werden können:

- `defined_in` $\subseteq$ `Component` $\times$ `Project`
  beschreibt die Projektzugehörigkeit einer Komponente, wie bereits in Bsp. 10.7 dargestellt.

- `includes` $\subseteq$ `Project` $\times$ `Project`
  beschreibt die Verfügbarkeit von Komponenten aus einer im Rahmen eines Projektes definierten Bibliothek innerhalb eines anderen Projektes.

- `contains` $\subseteq$ `Service` $\times$ `SIB` und
  `contains` $\subseteq$ `Service` $\times$ `Macro`
  dokumentieren den Aufbau eines Dienstes, d.h. welche Typen von *SIB*s und Makros in dem jeweiligen Dienst enthalten sind.

- `contains` $\subseteq$ `Macro` $\times$ `SIB` und
  `contains` $\subseteq$ `Macro` $\times$ `Macro`
  dokumentieren den Aufbau eines Makros, d.h. welche Typen von *SIB*s und anderen Makros in dem jeweiligen Makro enthalten sind.

- `calls` $\subseteq$ `Service` $\times$ `Service`
  dokumentiert, welche anderen Dienste innerhalb eines Dienstes in ihrer Ausführung angestoßen werden.

- `lies_in` $\subseteq$ `ContextKey` $\times$ `DataContext`
  legt fest, welche Variablen in den Datenkontexten der Applikation zur Verfügung stehen.

- `read` $\subseteq$ `Component` $\times$ `ContextKey` und
  `write` $\subseteq$ `Component` $\times$ `ContextKey`
  beschreiben, welche Variablen in den jeweiligen Datenkontexten gelesen bzw. geschrieben werden.

- `of_type` $\subseteq$ `ContextKey` $\times$ `Type`
  legt fest, von welchem Typ die Variablen sind, welche in den Datenkontexten der Applikation gehalten werden.

- `belongs_to` $\subseteq$ `Component` $\times$ `Domain` und
  `belongs_to` $\subseteq$ `BusinessObject` $\times$ `Domain`
  beschreiben die Einordnung von Komponenten und Geschäftsobjekten in die Teilgebiete des jeweiligen Anwendungsbereichs einer Applikation.

Kapitel 11

Personalisierungsframework

Web-basierte Applikationen gewinnen immer mehr an Bedeutung in den veschiedensten Bereichen des täglichen Lebens: Online-Shops, Online-Banking, Online-Auktionen, Online-Dienste kommunaler Einrichtungen sowie sonstiger Institutionen und Webportale von Firmen für ihre Kunden, sind nur einige Beispiele dafür.

Der Begriff Personalisierung ist dabei nicht einheitlich definiert. Man unterscheidet generell folgende drei Bereiche ([66, 69, 61]):

- rollen- und/oder rechte-basiertes System
- kundenspezifische Anpassungen (=customization)
- Analyse und Auswertung des Browsing-Verhaltens (=tracking)

Da die beiden letzteren Aspekte nur applikations-spezifisch realisierbar sind, beschränken wir uns im Rahmen dieser Arbeit auf den ersten Aspekt, die Realisierung eines rollen- und rechte-basierten Systems.

Dabei umfasst der Begriff Personalisierung – wie er im Rahmen dieser Arbeit verwendet wird – folgende Bereiche:

- Anpassung des Applikationsverhaltens sowie der enthaltenen Funktionalitäten an einzelne Benutzer und Benutzergruppen sowie deren Bedürfnisse, Berechtigungen und Verpflichtungen
- Einschränkung und Prüfung der Zugriffsberechtigung anhand der für einen Benutzer gespeicherten Rechte
- Anbieten von Navigationsmöglichkeiten sowie Präsentation von Inhalten für einen Benutzer basierend auf den Daten, die für diesen Benutzer gespeichert sind, d.h. Erzeugen persönlicher Sichten

11.1 Grundlegende Begriffe

In diesem Abschnitt werden einige grundlegende Begriffe für die in dieser Arbeit verwendete Art der Personalisierung definiert und in die in Kapitel 6 eingeführte Schichtendarstellung eingeordnet.

Abb. 11.1 zeigt hierfür eine etwas anschaulichere Darstellung des Schichtenmodells einer web-basierten Applikation aus Abb. 6.1. Dabei werden folgende Details explizit visualisiert:

- Benutzer einer Applikation,
- Features bzw. die sie realisierenden Dienste innerhalb der Koordinationsschicht einer Applikation,
- Geschäftsobjekte innerhalb der Datenschicht einer Applikation und
- Datenbanktabellen innerhalb der Persistenzschicht einer Applikation

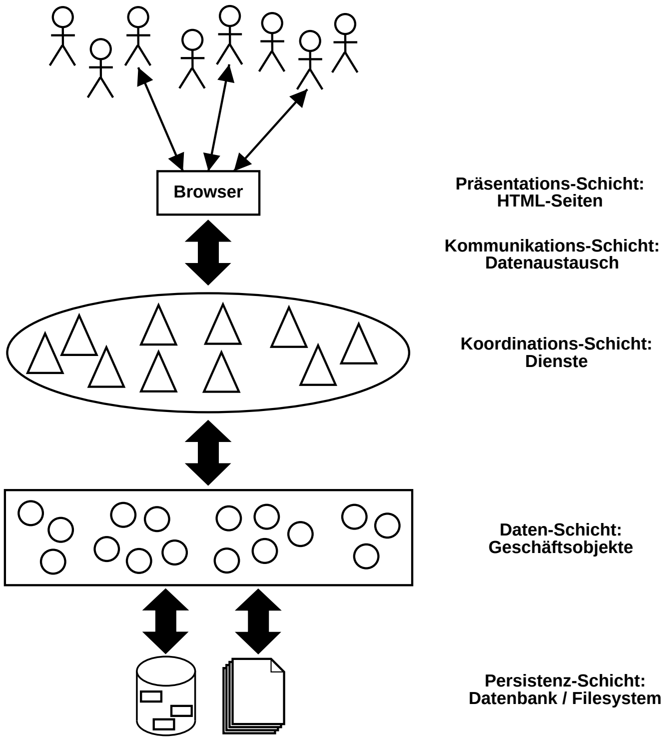


Abbildung 11.1: Schichten einer web-basierten Applikation

Dieses verfeinerte Schichtenmodell wird verwendet, um die Bedeutung der grundlegenden Begriffe im Rahmen der Personalisierung – wie sie innerhalb dieser Arbeit verwendet werden – zu veranschaulichen.

Abb. 11.2 führt diese verschiedenen Begriffe ein und stellt die Zusammenhänge und Beziehungen zwischen ihnen dar.

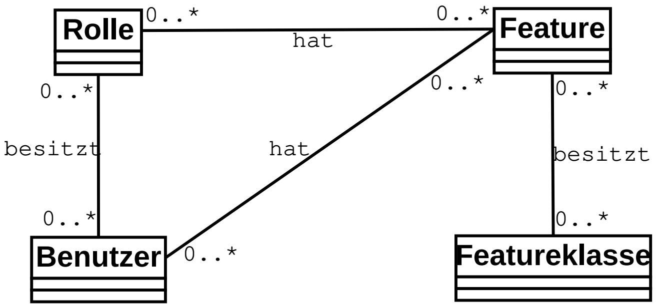


Abbildung 11.2: Begriffslandkarte – Personalisierung

Bei der Anforderungsdefinition einer Applikation mittels UML (vgl. [11]) können diese Begriffe mithilfe folgender Diagrammelemente in einem Anwendungsfalldiagramm dargestellt werden – wie auch in Abb. 11.3 skizziert:

- ***Benutzer*** bzw. homogene Gruppen von Benutzern interagieren mit einer Applikation und werden daher als Akteure modelliert.
- Die ***Features***, welche eine Applikation bereitstellt, werden als Anwendungsfälle modelliert. Features können sich aus externen Diensten und Utility-Diensten zusammensetzen (siehe Abschnitt 9.2). Externe Dienste werden als Anwendungsfälle dargestellt, die von einem Akteur ausgeführt werden können, Utility-Dienste dagegen sind immer über eine `<<include>>`-Beziehung an einen externen Dienst gekoppelt und damit nur indirekt erreichbar.
- Eine Gruppierung von Features gemäß ihres Ziels bzw. Verwendungszweckes wird als ***Featureklasse*** bezeichnet und als Paket dargestellt, welches die zu den Features gehörigen Anwendungsfälle beinhaltet.
- Ein ***Recht*** eines Benutzer bzw. einer homogenen Gruppe von Benutzern wird als Verbindung zwischen einem Akteur und einem Anwendungsfall dargestellt. Dies bedeutet, dass der Akteur den zugeordneten Anwendungsfall ausführen darf, d.h. die Berechtigung zum Ausführen dieses Anwendungsfalls besitzt.
- Der Begriff ***Rolle*** bezeichnet aus Benutzersicht eine homogene Gruppe von Benutzern, aus Featuresicht eine Menge von Features bzw. eine Menge von

Rechten. Demnach wird eine Rolle als Akteur modelliert und kann als Synonym zu dem Begriff Benutzergruppe (mit der ihr zugeordneten Menge von Features bzw. Rechten) verwendet werden.

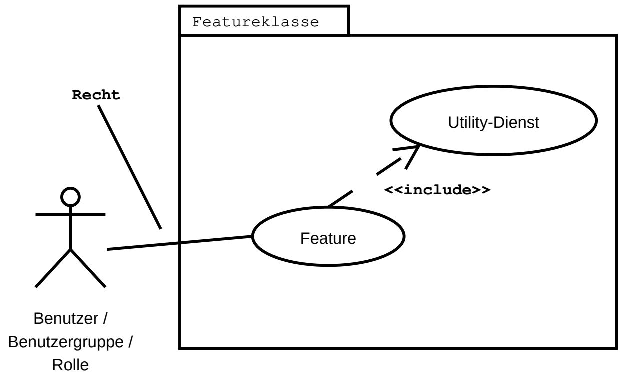


Abbildung 11.3: Personalisierungsbegriffe in UML

Eine detaillierte Beschreibung der einzelnen Begriffe ist in den folgenden Abschnitten enthalten.

11.1.1 Benutzer

Ein Benutzer ist eine Person der realen Welt, die mit einer web-basierten Applikation über einen Browser interagiert – wie in Abb. 11.4 dargestellt.

Daher werden Benutzer im Rahmen der Anforderungsdefinition mittels UML, d.h. im Anwendungsfalldiagramm, als Akteure modelliert (siehe Abb. 11.3).

Dabei unterscheiden wir zwischen Benutzern, welche anonym mit der Applikation interagieren, und solchen, die bei der Applikation registriert, d.h. bekannt sind. Von diesem Status eines Benutzers hängt es u.a. ab, welcher Anteil der von einer Applikation in Form von Diensten bereitgestellten Funktionalitäten (= Features, siehe Abschnitt 11.1.2) diesem Benutzer zur Verfügung steht.

Zugriff auf öffentliche (d.h. frei zugängliche) Features einer Applikation hat grundsätzlich jeder Benutzer. Eine Applikation kann jedoch auch private Features bereitstellen. Diese werden nur nach vorheriger Identifizierung (d.h. Anmeldung bei der Applikation) für einen Benutzer freigegeben, wenn dieser das Recht (siehe Abschnitt 11.1.4) hat, diese Features auszuführen.

Um diesen Identifizierungsprozess durchführen zu können sind im Rahmen des in diesem Kapitel beschriebenen Personalisierungsframeworks persönliche Daten eines Benutzers – mindestens die Accountdaten Benutzername und Passwort – innerhalb

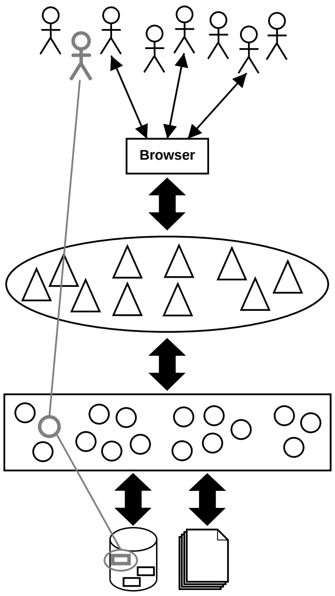


Abbildung 11.4: Personalisierung – Begriff *Benutzer*

einer web-basierten Applikation bekannt, d.h. in der zugehörigen Persistenzschicht gespeichert. Anhand dieser gespeicherten Daten kann sich ein Benutzer bei der Applikation anmelden und hat dann dadurch Zugriff auf die von der Applikation für ihn bereitgestellten privaten Features.

Benutzer, welche dieselben Rechte besitzen, nehmen dieselbe Rolle im Rahmen der Applikation ein und werden daher zu einer Benutzergruppe zusammengefasst (siehe Abschnitt 11.1.3).

11.1.2 Feature

Wie bereits in Kapitel 1 dargestellt, wird die Gesamtfunktionalität einer Applikation in Geschäftsprozesse unterteilt, an deren Ausführung i.d.R. verschiedene Personen bzw. Personengruppen beteiligt sind. Diese Geschäftsprozesse wiederum werden in Features aufgeteilt, an deren Ausführung jeweils nur eine Person oder Personengruppe beteiligt ist.

Jedes *Feature* wird dann mithilfe von Diensten realisiert (siehe Abb. 11.5). In der Regel wird eine Feature durch einen externen Dienst realisiert. In Einzelfällen kann bei der Definition eines Features unter dem Aspekt der Wiederverwendbarkeit – wie in Abschnitt 9.2 beschrieben – aber auch die Kombination eines externen Dienstes mit einer Menge von Utility-Diensten nötig und sinnvoll sein.

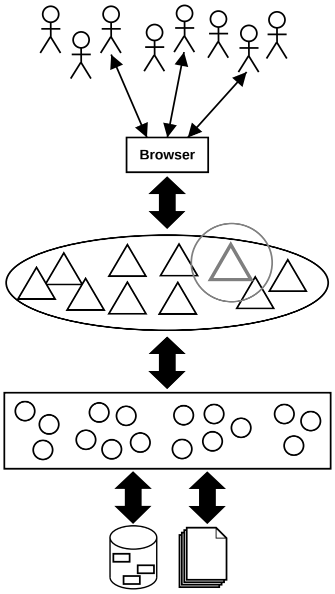


Abbildung 11.5: Personalisierung – Begriff *Feature*

Demnach wird ein Feature im Rahmen der Anforderungsdefinition mittels UML, d.h. im Anwendungsfalldiagramm, als Anwendungsfall oder als Kombination von Anwendungsfällen mittels `<<include>>`-Beziehung modelliert (siehe Abb. 11.3).

Darüber hinaus können zwei Features f_1 und f_2 miteinander in Beziehung stehen oder sich gegenseitig beeinflussen. Hierbei dient die Datenbank als globaler Zustandsraum für die Features.

Jedes Feature muss für sich alleine ausführbar sein, d.h. es muss jeweils geprüft werden, ob bzgl. der Datenbank und des aktuellen Benutzers alle notwendigen Voraussetzungen für das erfolgreiche Ausführen des jeweiligen Features erfüllt sind. Anderenfalls muss eine geeignete Fehlerbehandlung angestoßen werden.

Beispielsweise kann eine erfolgreiche Ausführung von f_2 nur dann möglich sein, wenn zuvor bereits f_1 erfolgreich ausgeführt worden ist. Solche Abhängigkeiten sind applikations-spezifisch, werden über das Datenmodell des Anwendungsbereiches festgelegt und müssen bei der Realisierung des Kontrollflusses der Applikation berücksichtigt und umgesetzt werden.

Beispiel 11.1:

Im Folgenden sind einige Features aus dem TEMPLUS Projekt beschrieben:

- Das Feature 'Anmeldestatistik' wird durch den Dienst *showGroupRegistrationStatistics* realisiert und liefert einem Organisator die Information, wie viele Studierende sich bereits für die Übungsgruppen einer ausgewählten Lehrveranstaltung angemeldet haben und wie beliebt die einzelnen Termine bei den Studierenden sind.

- Das Feature 'Daten einer Lehrveranstaltung anzeigen' setzt sich aus dem externen Dienste *viewCourse* und dem Utility-Dienst *mail* zusammen und steht Benutzern aller Rollen zur Verfügung. Dabei liefert der externe Dienst alle Daten der gewählten Lehrveranstaltung. Dies beinhaltet auch die Liste der Dozenten, Organisatoren und Tutoren, welche der Lehrveranstaltung zugeordnet sind. Diese Personen können dann mithilfe des E-Mail Utility-Dienstes kontaktiert werden.
- Die Features 'zu einer Lehrveranstaltung anmelden' und 'zu Gruppen anmelden' ermöglichen es Studierenden, sich für eine Lehrveranstaltung und deren zugehörige Gruppen als Teilnehmer anzumelden. Diese Features stehen in Beziehung zueinander und werden durch die Dienste *courseRegistrationOwn* bzw. *groupRegistrationOwn* realisiert. Dabei kann die Anmeldung zu den Gruppen erst erfolgen, wenn zuvor die Anmeldung zur zugehörigen Lehrveranstaltung erfolgreich durchgeführt worden ist. Zu Beginn des Features 'zu Gruppen anmelden' wird zunächst die Menge der Lehrveranstaltungen ermittelt, welche einer/m Studierenden angeboten werden. Eine Lehrveranstaltung erscheint nur in der Auswahlliste, falls der/die Studierende bereits als Teilnehmer für die Lehrveranstaltung angemeldet ist.

⌟

11.1.3 Rolle

Benutzer einer web-basierten Applikation haben unterschiedliche Aufgaben, Qualifikationen, Interessen und Bedürfnisse innerhalb des Anwendungsbereiches der Applikation.

Eine (homogene) *Benutzergruppe* wird definiert, indem man für eine Menge gleichartiger bzw. -berechtigter Benutzer einen Namen vergibt.

Wird eine Benutzergruppe mit einer Menge von Features assoziiert (siehe Abb. 11.6), bezeichnen wir dies als *Rolle*.

Ein Benutzer hat eine Rolle inne, wenn er Mitglied der zugehörigen Benutzergruppe ist und dementsprechend alle mit dieser Benutzergruppe assoziierten Features ausführen darf. Dies bedeutet, dass ein Benutzer, der Mitglied einer Benutzergruppe ist, das Recht hat, diejenigen Features auszuführen, die mit dieser Benutzergruppe assoziiert sind.

Demnach wird eine Rolle im Rahmen der Anforderungsdefinition mittels UML, d.h. im Anwendungsfalldiagramm, als Akteur modelliert (siehe Abb. 11.3).

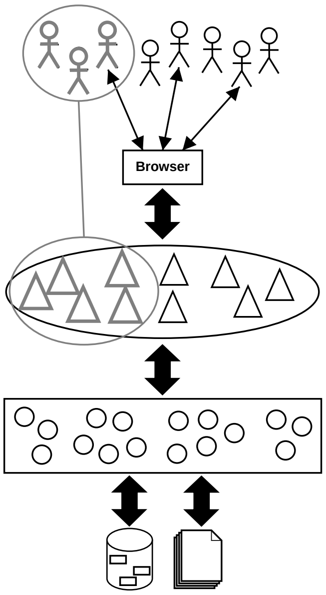


Abbildung 11.6: Personalisierung – Begriff *Rolle*

Beispiel 11.2:

Im universitären Alltag gibt es beispielsweise die Benutzergruppen bzw. Rollen *Student*, *Dozent*, *Tutor*, *Organisator*. Diese verschiedenen Benutzergruppen haben unterschiedliche Aufgaben und Interessen im Rahmen des Lehremanagements an einer Universität, dem Anwendungsbereich der TEMPLUS Applikation.

Folgende Aufgaben können die genannten Benutzergruppen bei der TEMPLUS Applikation z.B. in Bezug auf Geschäftsobjekte vom Typ Lehrveranstaltung wahrnehmen:

- *Studenten* nehmen an Lehrveranstaltungen teil,
- *Dozenten* bieten Lehrveranstaltungen an und führen diese durch,
- *Tutoren* betreuen praktische Übungen, die an eine Lehrveranstaltung gekoppelt sind, und
- *Organisatoren* sind mit der Organisation von Lehrveranstaltungen und den zugehörigen praktischen Übungen betraut.

⌟

11.1.4 Recht

Ein *Recht* ist die Erlaubnis, ein Feature auszuführen, und kann sowohl Benutzergruppen (siehe Abb. 11.6) als auch einzelnen Benutzern zugeordnet werden.

Ein Benutzer hat das Recht ein Feature auszuführen, wenn dieses Feature entweder direkt mit diesem Benutzer assoziiert ist oder der Benutzer Mitglied einer Benutzergruppe ist, mit der das Feature assoziiert ist.

Ein Recht wird daher im Rahmen der Anforderungsdefinition mittels UML, d.h. im Anwendungsfalldiagramm, als Verbindung zwischen einem Akteur und einem Anwendungsfall modelliert (siehe Abb. 11.3).

11.1.5 Featureklasse

Komplexe Applikationen können aufgrund des Umfangs ihrer Gesamtfunktionalität aus sehr vielen Features bestehen. Je größer dieser Menge der Features wird, desto mehr schwinden Handhabbarkeit und Übersichtlichkeit.

Insbesondere innerhalb der Navigation, über welche den Benutzern die von der Applikation zur Verfügung gestellten Features angeboten werden, kann dies zu Problemen führen.

Aus diesem Grund werden Features, die überwiegend auf denselben Typen von Geschäftsobjekten bzw. Daten arbeiten, zu einer *Featureklasse* (siehe Abb. 11.7) gruppiert, die einen Bereich innerhalb des Anwendungsspektrums der Applikation beschreibt. Dadurch erzielt man eine Strukturierung der Gesamtfunktionalität einer Applikation.

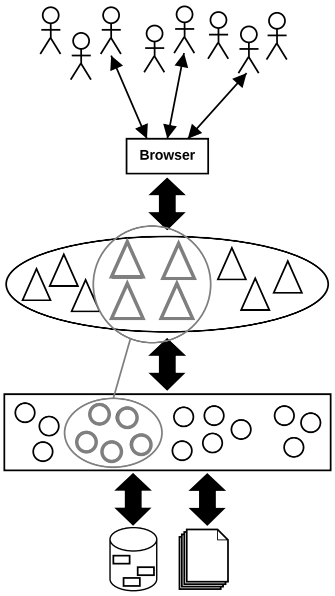


Abbildung 11.7: Personalisierung – Begriff *Featureklasse*

Eine Featureklasse wird im Rahmen der Anforderungsdefinition mittels UML, d.h. im Anwendungsfalldiagramm, als Paket modelliert, welches die gruppierten Features – dargestellt als Anwendungsfälle – beinhaltet (siehe Abb. 11.3 bzw. 11.8).

Beispiel 11.3:

Innerhalb der TeMPlUs Applikation sind die Features

- 'Lehrveranstaltung anlegen',
- 'Daten einer Lehrveranstaltung modifizieren',
- 'Daten einer Lehrveranstaltung anzeigen',
- 'zu einer Lehrveranstaltung anmelden' und
- 'Teilnehmerliste einer Lehrveranstaltung anzeigen'

der Featureklasse *Lehrveranstaltung* zugeordnet, da all diese Features überwiegend auf Objekten vom Typ `Course`[1] arbeiten.

Abb. 11.8 zeigt den hier im Beispiel beschriebenen Ausschnitt aus der Featureklasse *Lehrveranstaltung* in Form eines UML-Anwendungsfalldiagramms.

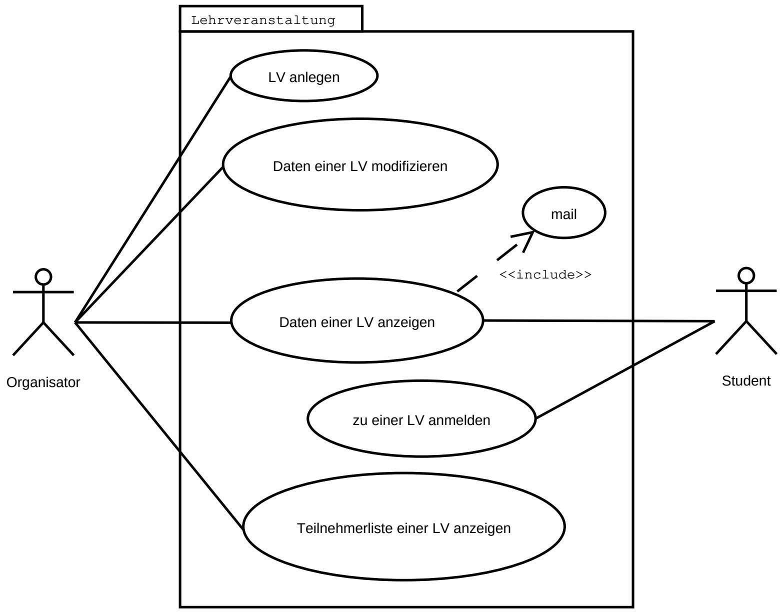


Abbildung 11.8: Featureklasse *Lehrveranstaltung* im Anwendungsfalldiagramm

⌟

[1]Die Klasse `Course` stellt innerhalb des TeMPlUs Projektes eine Spezifikation für Geschäftsobjekte vom Typ 'Lehrveranstaltung' bereit.

11.2 Realisierung

Unter Verwendung der in Abschnitt 11.1 eingeführten grundlegenden Begriffe wird nun hier die Realisierung des flexibel konfigurierbaren und einsetzbaren Personalisierungsframeworks präsentiert.

- Abschnitt 11.2.1 gibt einen Überblick über die *ABC-SD*-Projekte, welche die Realisierung des Personalisierungsframeworks beinhalten. Die einzelnen Projekte werden in den Abschnitten Abschnitt 11.2.2 bis Abschnitt 11.2.5 detaillierter beschrieben.

- Die durch das Personalisierungsframework bereitgestellten Basisdienste, welche als wiederverwendbare Komponenten für die Entwicklung personalisierter, web-basierter und als Dienstfamilie konzipierter Applikationen mit *ABC-SD* zur Verfügung stehen, werden im Rahmen der Projektbeschreibung jeweils kurz erwähnt und dann in Abschnitt 11.3 im Detail präsentiert.

11.2.1 Projektübersicht

Die Realisierung des Personalisierungsframeworks ist auf verschiedene Projekte im Rahmen des *ABC-SD* aufgeteilt, welche aufeinander aufbauen. Abb. 11.9 zeigt die gesamte Projekthierarchie im Überblick.

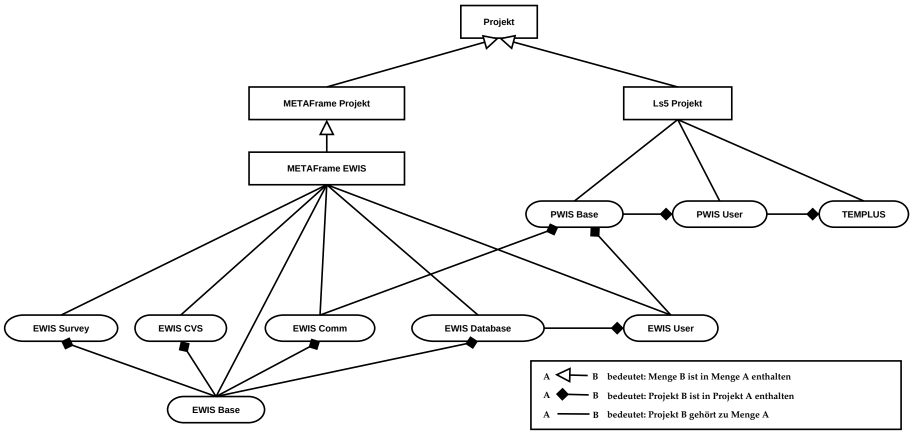


Abbildung 11.9: Die Hierarchie aller verwendeten *ABC-SD*-Projekte

- Ausgangspunkt und Basis für die Entwicklung dieses Personalisierungsframeworks ist das im MetaFrame Projekt *EWIS* User ([48]) umgesetzte Benutzer- und Rollenmanagement, welches in das *ABC-SD* integriert ist. Es dient zur

Verwaltung von Benutzern und Rollen sowie deren Zuordnung zueinander und ist in Abschnitt 11.2.2 näher beschrieben.

- Das PwisBase Projekt (siehe Abschnitt 11.2.3) basiert auf dem *EWIS* User Projekt und beinhaltet die Erweiterung des Benutzer- und Rollenmanagements um ein datenbank-basiertes Feature-Management, welches die Verwaltung der von einer Applikation bereitgestellten Rollen, Featureklassen und Features sowie die Beziehung letzterer zu den Rollen und Featureklassen realisiert. Die hierfür benötigte Funktionalität wird über eine eigenständige, web-basierte Applikation bereitgestellt.

  Zusätzlich wird eine Behandlung von Rollendaten ermöglicht, wobei für jeden einzelnen Rollentyp zusätzliche, rollen-spezifische Informationen (wie z.B. Matrikelnummer bei der Rolle *Student* oder Sprechzeiten bei der Rolle *Dozent*) gespeichert werden können.

- Das PwisUser Projekt (siehe Abschnitt 11.2.4) erweitert das PwisBase Projekt um eine Benutzer-Klasse. Diese ermöglicht es – neben den bereits durch das Projekt *EWIS* User vorgegebenen Accountdaten – auch benutzer-spezifische Daten ähnlich wie in einem gängigen Kontaktmanagement-Tool (vgl. [90, 89]) zu verwalten, d.h. Titel, Vorname, Nachname, Geburtsdatum, mehrere E-Mail-Adressen, mehrere Adressen und mehrere Telefonnummern.

  Unter Verwendung dieser Benutzerklasse werden im Rahmen dieses Projektes Basisdienste als wiederverwendbare Komponenten realisiert, welche die Verwaltung von Benutzern und deren Rollen realisieren und darüber hinaus Navigationsfunktionalität für personalisierte, web-basierte Applikationen bereitstellen.

- Das TeMPLUs Projekt (siehe Abschnitt 11.2.5) ist schließlich ein konkretes Anwendungsbeispiel für das durch das Projekt PwisUser definierte Personalisierungsframework. Insbesondere wird hier exemplarisch anhand der Rolle *Student* die Spezifikation von Rollendaten und deren Einordnung in die durch das bestehende PwisUser Projekt vorgegebene Klassenstruktur illustriert.

Die folgenden Abschnitte enthalten eine detailliertere Beschreibung der einzelnen Projekte sowie der Zusammenhänge und Abhängigkeiten zwischen diesen.

11.2.2 Das *EWIS* User Projekt

Das *EWIS* User Projekt beinhaltet ein Benutzer- und Rollenmanagement, d.h. es realisiert die für die Verwaltung von Benutzern und Benutzergruppen (Rollen) sowie der Mitgliedschaft von Benutzern in Benutzergruppen nötigen Geschäftsobjekte und stellt *SIB*s zur Verfügung, welche auf diesen Geschäftsobjekten arbeiten. Zusätzlich zu den verschiedenen Interfaces, über die die Schnittstellen und Zusammenhänge der Geschäftsobjekte spezifiziert werden, bietet dieses Projekt auch JDBC-basierte[2] Standardimplementierungen dieser Interfaces an.

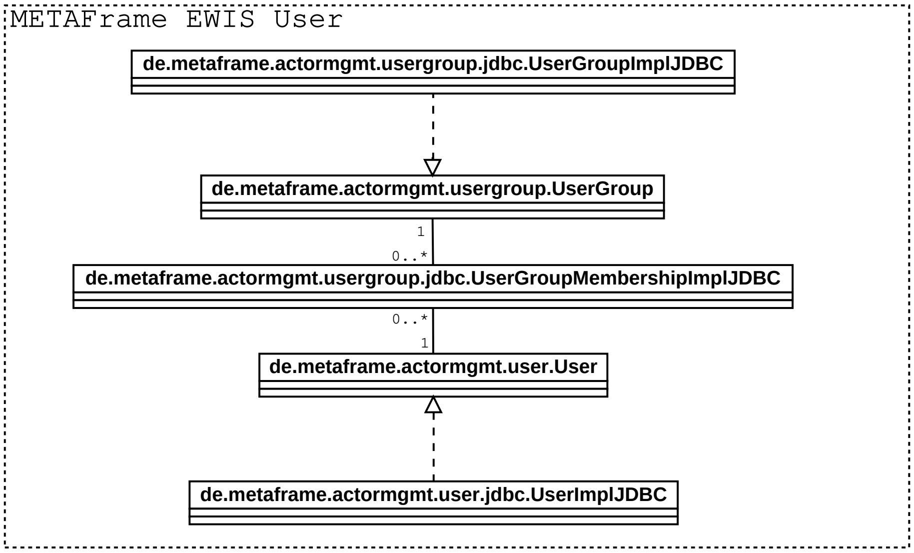


Abbildung 11.10: Benutzer-Rollen-Management im *EWIS* User Projekt

Abb. 11.10 illustriert den wesentlichen Ausschnitt des Benutzer- und Rollenmanagements anhand eines UML Klassendiagramms. Die dort verwendeten Klassen und Interfaces werden im Folgenden genauer erläutert:

- Das Interface `User` bestimmt die Schnittstellen für Objekte, die Benutzer (siehe Abschnitt 11.1.1) innerhalb der Applikation repräsentieren – wie in Anhang A.1.1 beschrieben. Das einem Benutzer zugeordnete `User` Objekt stellt dabei dessen Benutzeraccount (mit eindeutiger Id, Benutzername, Passwort) innerhalb einer personalisierten, web-basierten Applikation dar.

- Die Klasse `UserImplJDBC` ist eine JDBC-basierte Standardimplementierung des Interfaces `User`.

- Das Interface `UserGroup` legt die Schnittstellen für Objekte fest, die Benutzergruppen (siehe Abschnitt 11.1.3) innerhalb einer Applikation repräsentieren. Hierfür deklariert das Interface Methoden, die alle Objekte vom Typ

[2]JDBC steht für Java Database Connectivity und ist eine standardisierte Datenbankschnittstelle für Java. Diese Technologie erlaubt es, das entwickelte Benutzer- und Rollenmanagement mit einer beliebigen SQL-Datenbank zu verwenden, die einen JDBC-Treiber zur Verfügung stellt.

`UserGroup` gemeinsam haben (siehe Anhang A.1.2). Ein Objekt vom Typ `UserGroup` legt also einfach einen Namen für eine homogene Menge von Objekten vom Typ `User` fest. Im Rahmen personalisierter, web-basierter Applikationen betrachten wir – wie in Abschnitt 11.1.3 beschrieben – eine Benutzergruppe als Rolle, welche die Mitglieder dieser Benutzergruppe innehaben.

- Die Klasse `UserGroupImplJDBC` ist eine JDBC-basierte Standardimplementierung des Interfaces `UserGroup`.
- Die Klasse `UserGroupMembershipImplJDBC` ist eine JDBC-basierte Standardimplementierung des im Anhang A.1.3 beschriebenen Interfaces `UserGroupMembership`. Das Interface `UserGroupMembership` (siehe Anhang A.1.3) deklariert Methoden, die alle Klassen gemeinsam haben, die eine Assoziation zwischen Benutzern und Benutzergruppen – d.h. zwischen Objekten vom Typ `User` und Objekten vom Typ `UserGroup` – repräsentieren. Die Klasse `UserGroupMembershipImplJDBC`, welche die Mitgliedschaft eines Benutzers in einer Benutzergruppe realisiert, implementiert dieses Interface.

11.2.3 Das PwisBase Projekt

Das PwisBase Projekt erweitert das *EWIS* User Projekt um folgende Aspekte:

- Es wird eine datenbank-basierte Verwaltung der Features einer web-basierten Applikation realisiert.
- Darüber hinaus besteht die Möglichkeit zur Verwaltung verschiedener Typen von Rollendaten für unterschiedliche Rollen.
- Für die Definition der Features, Rollen und Featureklassen, die Zuordnung der Features zu Featureklassen sowie die Konfiguration der Rechte der verschiedenen Benutzergruppen bietet das PwisBase Projekt entsprechende Administrationsfunktionalität in Form von Basisdiensten über eine eigenständige, web-basierte Applikation an, welche in Abschnitt 11.3.1 näher beschrieben wird.

Dieses Feature-Management ermöglicht die Konfiguration und Verwaltung der Features und Featureklassen einer personalisierten, web-basierten Applikation sowie der Rechte, die in diesem Rahmen an Benutzergruppen vergeben werden können.

Da die Verwaltung der Features über die Datenbank erfolgt, können zusätzliche Informationen (z.B. Name, Beschreibung, URL des zu dem Feature gehörigen externen Dienstes, etc.) für die einzelnen Features gespeichert und im Rahmen der Applikation weiter genutzt werden.

Abb. 11.11 illustriert die Erweiterung des Benutzer- und Rollenmanagements aus dem *EWIS* User Projekt anhand eines UML Klassendiagramms. Die verwendeten zusätzlichen Klassen werden im Folgenden genauer erläutert:

- Die Klasse `FeatureEntity` repräsentiert ein Feature (siehe Abschnitt 11.1.2) einer personalisierten, web-basierten Applikation im System. Eine Liste der Attribute dieser Klasse ist in Anhang A.2.1 zu finden.
- Die Klasse `FeatureClass` repräsentiert eine Featureklasse (siehe Abschnitt 11.1.5) innerhalb einer Applikation und ermöglicht so die logische Gruppierung von Features anhand des Datenmodells der Applikation. Eine Liste der Attribute dieser Klasse ist in Anhang A.2.2 zu finden.
- Die Klasse `FeatureEntityClassification` realisiert die Zuordnung von Features zu Featureklassen, d.h. sie assoziiert Objekte vom Typ `FeatureEntity` mit Objekten vom Typ `FeatureClass`. Eine Liste der Attribute dieser Klasse ist in Anhang A.2.3 zu finden.
- Die Klasse `WisUserGroupImplJDBC` repräsentiert eine Benutzergruppe der Applikation und ist eine Adaption der Klasse `UserGroupImplJDBC` aus dem Projekt *EWIS* User (siehe Abschnitt 11.2.2) an das Feature-Management, welches innerhalb des PwisBase Projektes definiert wird. Sie stellt dieselben Attribute wie die Klasse `UserGroupImplJDBC` zur Verfügung und realisiert darüber hinaus die Koppelung des bestehenden Benutzer- und Rollenmanagements an das neue Feature-Management.
- Die Klasse `WisUserImplJDBC` repräsentiert einen Benutzer der Applikation und ist eine Adaption der Klasse `UserImplJDBC` aus dem Projekt *EWIS* User (siehe Abschnitt 11.2.2) an das Feature-Management, welches innerhalb des PwisBase Projektes definiert wird. Sie stellt dieselben Attribute wie die Klasse `UserImplJDBC` zur Verfügung und realisiert darüber hinaus die Koppelung des bestehenden Benutzer- und Rollenmanagements an das neue Feature-Management.
- Die Klasse `FeatureRoleAssociation` realisiert die Zuordnung von Features zu Rollen, d.h. sie assoziiert Objekte vom Typ `FeatureEntity` mit Objekten vom Typ `WisUserGroupImplJDBC` und ermöglicht so die Definition der Rechte, die eine Benutzergruppe innehat. Eine Liste der Attribute dieser Klasse ist in Anhang A.2.4 zu finden.
- Die Klasse `UserFeatureAssociation` realisiert die Zuordnung von Features zu Benutzern, d.h. sie assoziiert Objekte vom Typ `FeatureEntity` mit Objekten vom Typ `WisUserImplJDBC` und ermöglicht so die Definition der Rechte, die ein Benutzer direkt innehat. Eine Liste der Attribute dieser Klasse ist in Anhang A.2.5 zu finden.

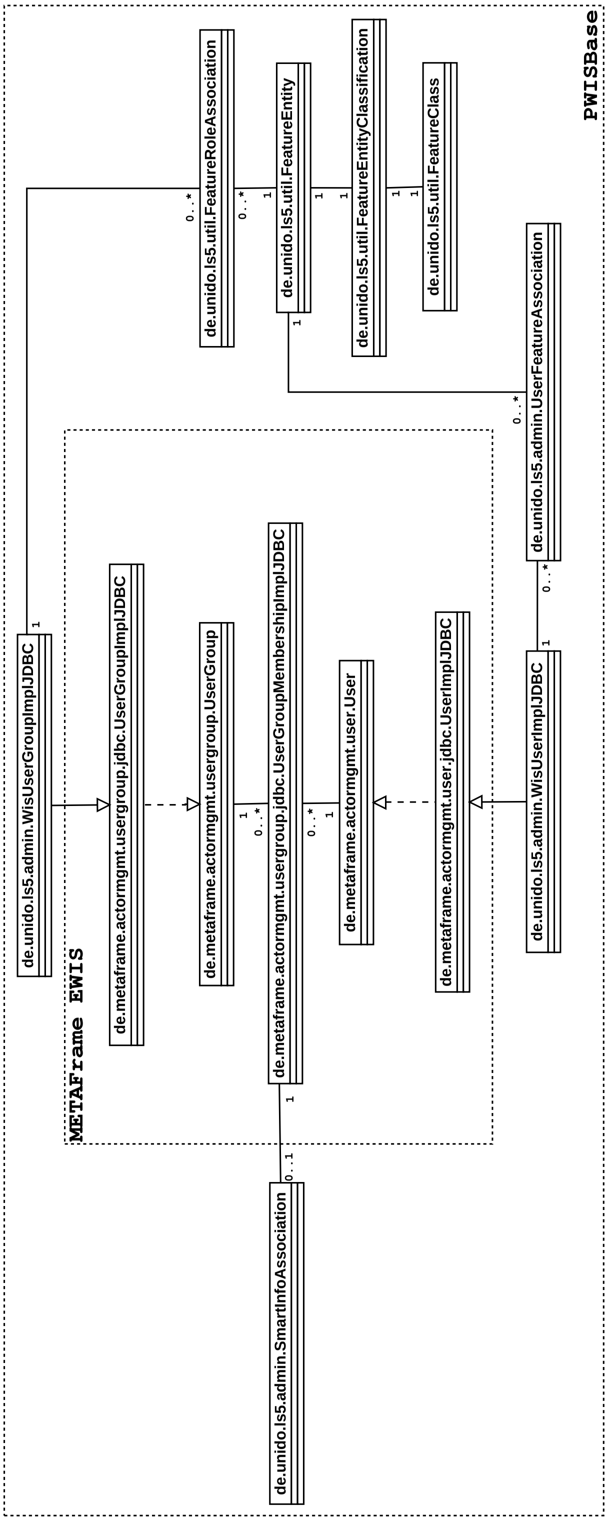


Abbildung 11.11: Benutzer-Rollen-Feature-Management im PwisBase Projekt

- Die Klasse `SmartInfoAssociation` ermöglicht es, applikations-spezifische Rollendaten als zusätzliche Information im Rahmen der Mitgliedschaft in einer Benutzergruppe zu speichern und zu verwalten, d.h. Objekte vom Typ `UserGroupMembershipImplJDBC` werden mit Geschäftsobjekten assoziert, welche entsprechende zu der jeweiligen Rolle gehörige applikations-spezifische Rollendaten repräsentieren. Dabei können verschiedene Rollen auch Rollendaten unterschiedlichen Typs haben. Eine Liste der Attribute dieser Klasse ist in Anhang A.2.6 zu finden.

11.2.4 Das PwisUser Projekt

Das PwisUser Projekt erweitert das PwisBase Projekt

- um die Realisierung zur Verwaltung von Benutzerdaten, die es erlaubt, neben den Kennungsdaten eines Benutzers auch Kontaktinformationen wie beispielsweise Titel, Name, Geburtsdatum, mehrere Adressen, mehrere E-Mail-Adressen und mehrere Telefonnummern (vgl. diverse Office-Anwendungen [90, 89]), sowie einige interne Informationen (z.B. Status eines Benutzeraccounts) zu speichern (siehe Abb. 11.12 und Abb. 11.13).

- Darüber hinaus werden in diesem Projekt einige Basis-Dienste für das Benutzer- und Rollenmanagement sowie Navigationsfunktionalität für personalisierte, web-basierte Applikationen definiert, die direkt bzw. nach geringfügigen Anpassungen in eine neue Applikation übernommen werden können (siehe Abschnitt 11.3.2).

Abb. 11.12 illustriert die Erweiterung des Feature-, Benutzer- und Rollenmanagements aus dem PwisBase Projekt anhand eines UML Klassendiagramms. Die verwendeten zusätzlichen Klassen werden im Folgenden genauer erläutert:

- Die Klasse `PWISUser` ermöglicht die Speicherung von Benutzerdaten ähnlich wie in einem gängigen Kontaktmanagement-Tools ([90, 89]). Eine Liste der Attribute dieser Klasse ist in Anhang A.3.1 zu finden. Direkte Oberklasse der Klasse `PWISUser` ist die Klasse `WisUserImplJDBC`.

- Die Klasse `UserAddress` enthält die Adressdaten eines Benutzers und wird mit dem zugehörigen Objekt vom Typ `PWISUser` assoziiert. Wir unterscheiden die Adresstypen `OFFICE` und `PRIVATE`. Jedem Benutzer können mehrere Adressen beliebigen Typs zugeordnet werden, eine muss jedoch als primär gekennzeichnet werden.

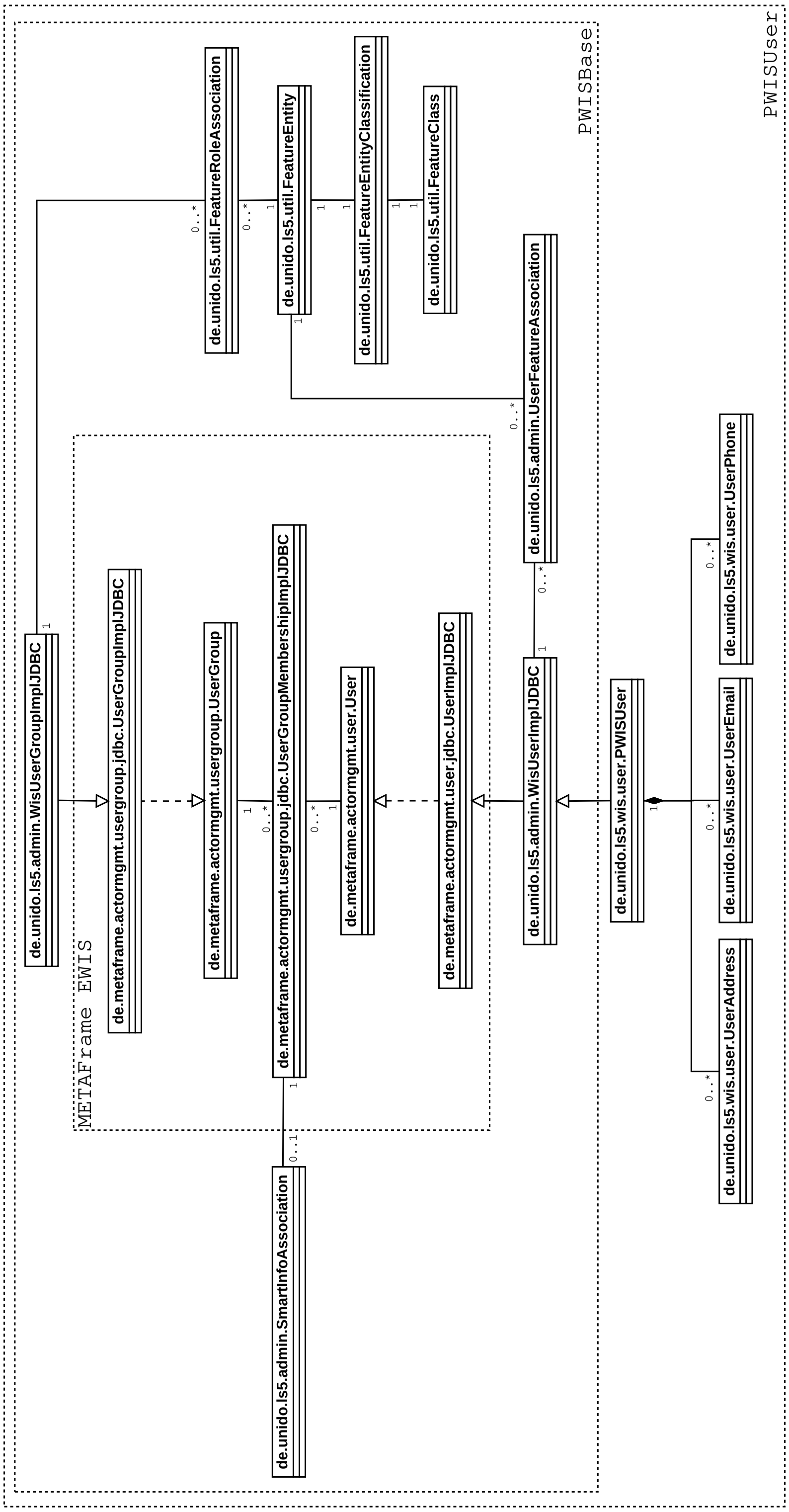


Abbildung 11.12: Benutzer-Rollen-Feature-Management im PwisUser Projekt

- Die Klasse `UserEmail` enthält die E-Mail-Adresse eines Benutzers und wird mit dem zugehörigen Objekt vom Typ `PWISUser` assoziiert. Wir unterscheiden die E-Mail-Adresstypen `OFFICE` und `PRIVATE`. Jedem Benutzer können mehrere E-Mail-Adressen beliebigen Typs zugeordnet werden, eine muss jedoch als primär gekennzeichnet werden.

- Die Klasse `UserPhone` enthält die Telefonnummer eines Benutzers und wird mit dem zugehörigen Objekt vom Typ `PWISUser` assoziiert. Wir unterscheiden die Telefonnummertypen `MOBILE`, `PRIVATE`, `OFFICE`, `PAGER`, `SECRETARIAT` und `FAX`. Jedem Benutzer können mehrere Telefonnummern beliebigen Typs zugeordnet werden, eine muss jedoch als primär gekennzeichnet werden.

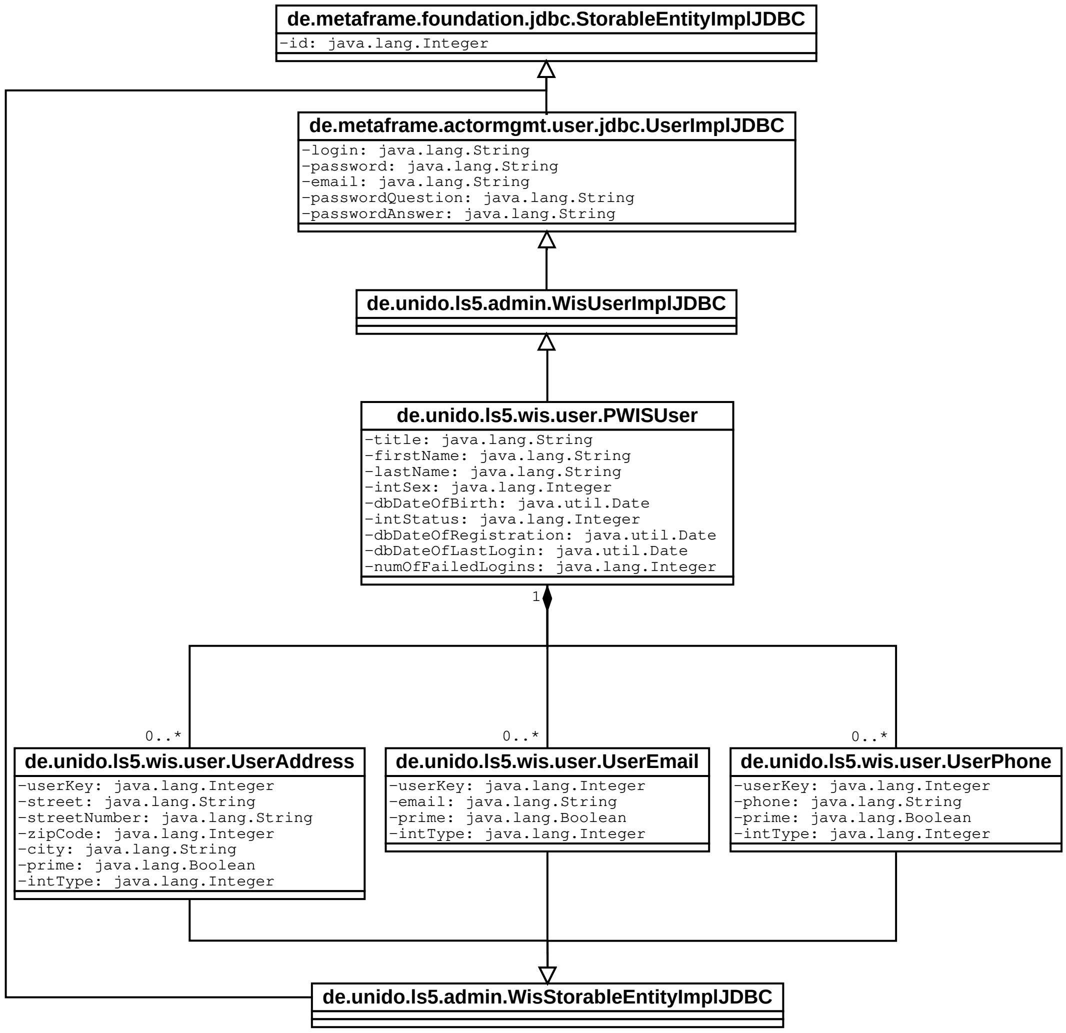


Abbildung 11.13: Beispiel einer applikations-spezifischen Benutzerklasse

11.2.5 Das TeMPlUs Projekt

Das TeMPlUs Projekt ist ein Beispiel einer personalisierten, web-basierten Applikation, die auf dem Projekt PwisUser aufsetzt, d.h. das darüber bereitgestellte Personalisierungsframework einsetzt. Die TeMPlUs Applikation unterstützt die Organisation und Durchführung von Lehrveranstaltungen an einer Universität, wie bereits in Abschnitt 2.2 beschrieben.

Bei einer neuen personalisierten, web-basierten Applikation, welche das Personalisierungsframework benutzt, müssen zunächst die nötigen Vorbereitungen getroffen werden, auf die in Kapitel 12 noch näher eingegangen wird. Dies umfasst unter anderem die Realisierung der benötigten applikations-spezifischen Rollendaten, welche dann unter Verwendung der Klasse `SmartInfoAssociation` als Zusatzinformation einer Rollenzugehörigkeit verwaltet werden können (siehe Abschnitt 11.2.3).

Abb. 11.14 zeigt dieses Szenario für eine beliebige Applikation.

Das TeMPlUs Projekt verwendet die Benutzerklasse, die im PwisUser Projekt (siehe Abschnitt 11.2.4) definiert ist. Darüber hinaus werden folgende Rollendaten definiert:

Die Klasse `StudentData` stellt ein Beispiel für Rollendaten dar. Sie realisiert die zusätzlichen Daten, die ein Benutzer mit Rolle *Student* besitzt, d.h. Matrikelnummer, Semester und Studiengang. Eine Liste der Attribute dieser Klasse ist in Anhang A.3.2 zu finden.

11.3 Funktionalitäten des entwickelten Personalisierungsframeworks

Basierend auf den in Abschnitt 11.1 eingeführten grundlegenden Begriffen für den Bereich Personalisierung – wie sie im Rahmen dieser Arbeit verwendet werden – geben die beiden folgenden Abschnitte einen Überblick über die Menge der Funktionalitäten, welche das entwickelte und in Abschnitt 11.2 detailliert beschriebene Personalisierungsframework abdeckt.

- **Administrationsfunktionalität**
  dient zur Domänenmodellierung für eine personalisierte, web-basierte Applikation und kommt während der Entwicklung dieser Applikation – d.h. *vor dem Bereitstellen* – zum Einsatz, wie in Abschnitt 11.3.1 beschrieben. Dabei werden die grundlegenden Begriffe aus dem Bereich der Personalisierung für eine Applikation festgelegt, d.h. Rollen, Features und Featureklassen sowie deren Beziehungen zueinander werden definiert.

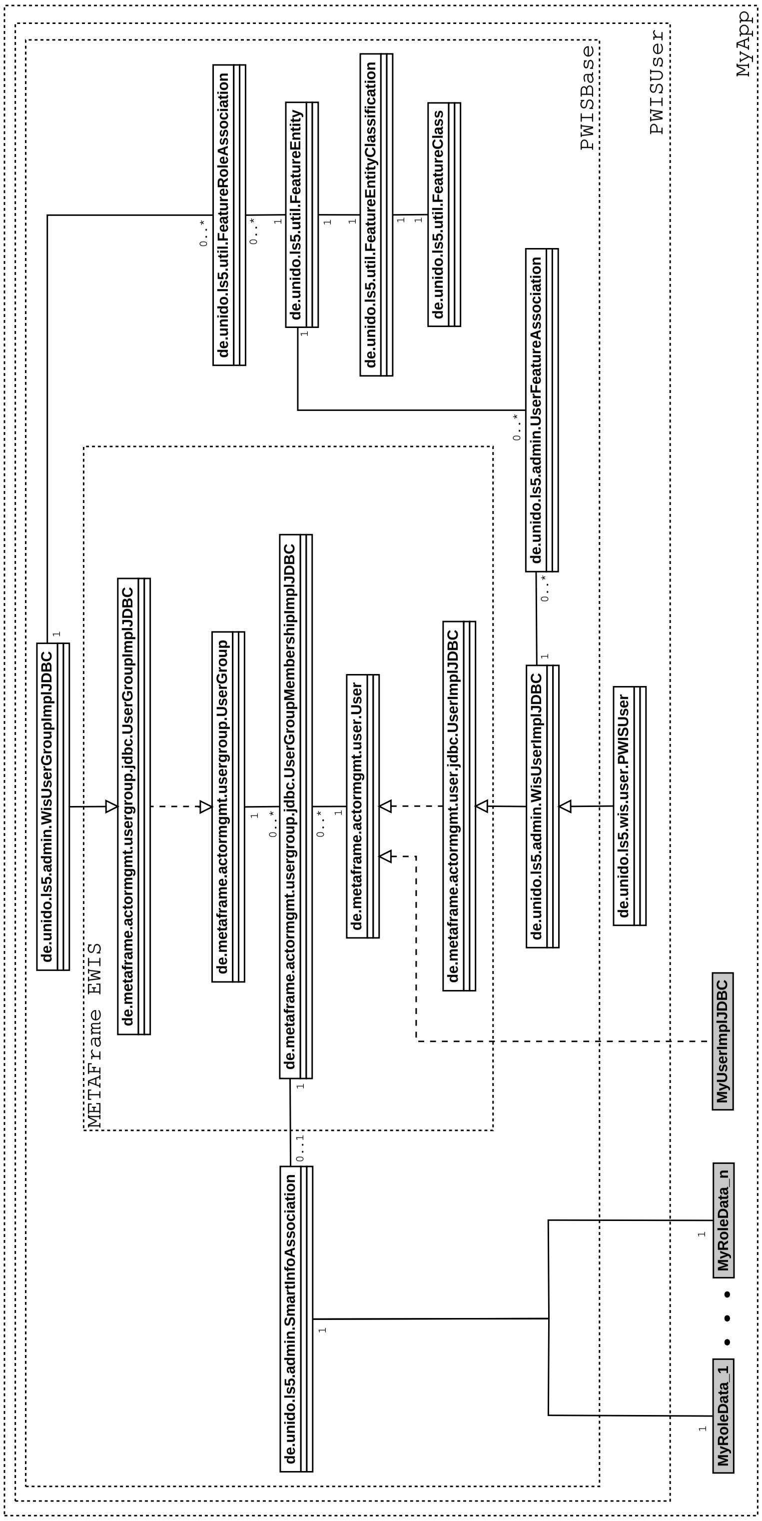


Abbildung 11.14: Benutzer-Rollen-Feature-Management allgemein

- **Verwaltungs- und Navigationsfunktionalität**
  bilden einen Teil der Gesamtfunktionalität einer personalisierten, web-basierten Applikation und realisieren die Verwaltung von Benutzern und deren Rechten sowie die grundlegenden Navigationsstrukturen innerhalb der Applikation. Diese Features werden mithilfe vordefinierter, wiederverwendbarer Dienste aus dem Personalisierungsframework umgesetzt und kommen erst zur Laufzeit einer Applikation – d.h. *nach dem Bereitstellen* – zum Einsatz, wie in Abschnitt 11.3.2 dargestellt.

All diese Funktionalitäten werden in Form von web-basierten Diensten als wiederverwendbare Komponenten für die Entwicklung personalisierter, web-basierter Applikationen realisiert.

11.3.1 Administrationsfunktionalität

Die Administrationsfunktionalität wird zur Konfiguration des Personalisierungsframeworks, d.h. zur Domänenmodellierung, für eine personalisierte, web-basierte Applikation eingesetzt.

Im Einzelnen werden dabei – wie bereits in Abschnitt 11.1 beschrieben – die grundlegenden Begriffe für den Bereich der Personalisierung im Rahmen einer personalisierten, web-basierten Applikation festgelegt, d.h.

- welche Features die Applikation bereitstellt,
- welche Featureklassen für den Anwendungsbereich der Applikation zur Verfügung stehen sollen,
- welche Rollen von Benutzern die Applikation unterstützt,
- welchen Featureklassen die einzelnen Features zugeordnet sind und
- mit welchen Rollen die einzelnen Features assoziiert werden.

Für diesen Zweck steht eine separate web-basierte Konfigurationsapplikation (siehe Abschnitt 12.3) zur Verfügung, welche diese Administrationsfunktionalität bereitstellt und mithilfe derer die Domänenmodellierung für die neu zu entwickelnde, personalisierte, web-basierte Applikation vorgenommen werden kann.

Die Administrationsfunktionalität wird mithilfe folgender Basisdienste realisiert. Abb. 11.15 liefert einen Überblick über diese Basisdienste anhand eines UML ([4, 5, 24]) Anwendungsfalldiagramms.

- Der Dienst *addFeature* dient zum Anlegen eines neuen Features innerhalb der Domäne der zu entwickelnden personalisierten, web-basierten Applikation. Dabei werden u.a. der Name und der Typ des Features sowie die URL des zugehörigen externen Dienstes festgelegt, über welche die Ausführung des Features angestoßen werden kann.

- Der Dienst *modifyFeature* erlaubt das Ändern der Daten bzw. Eigenschaften eines Features.

- Mithilfe des Dienstes *deleteFeature* kann ein Feature aus der Domäne einer web-basierten Applikation gelöscht werden.

- Der Dienst *showFeatures* zeigt alle Features an, welche innerhalb der Domäne einer personalisierten, web-basierten Applikation zur Verfügung stehen.

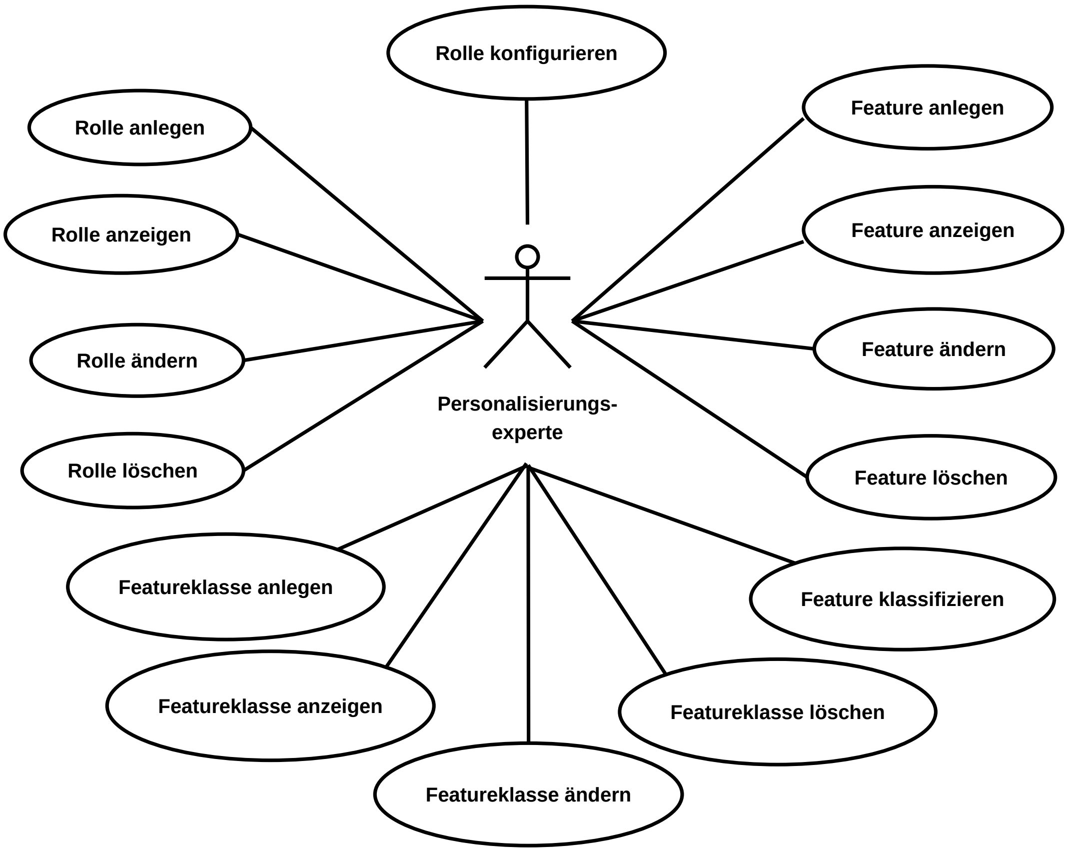


Abbildung 11.15: Administrationsfunktionalität

- Der Dienst *addFeatureClass* dient zum Anlegen einer neuen Featureklasse innerhalb der Domäne der zu entwickelnden personalisierten, web-basierten Applikation. Dabei werden u.a. der Name der Featureklasse sowie die URL des zugehörigen externen Dienstes festgelegt, welcher bei Auswahl der Featureklasse ausgeführt werden soll.

- Der Dienst *modifyFeatureClass* erlaubt das Ändern der Daten bzw. Eigenschaften einer Featureklasse.

- Mithilfe des Dienstes *deleteFeatureClass* kann eine Featureklasse aus der Domäne einer web-basierten Applikation gelöscht werden.

- Der Dienst *showFeatureClasses* zeigt alle Featureklassen an, welche innerhalb der Domäne einer personalisierten, web-basierten Applikation zur Verfügung stehen.

- Der Dienst *addRole* dient zum Anlegen einer neuen Rolle innerhalb der Domäne der zu entwickelnden personalisierten, web-basierten Applikation. Dabei werden u.a. der Name der Rolle sowie die URL des zugehörigen externen Dienstes festgelegt, welcher bei Auswahl der Rolle ausgeführt werden soll.

- Der Dienst *modifyRole* erlaubt das Ändern der Daten bzw. Eigenschaften einer Rolle.

- Mithilfe des Dienstes *deleteRole* kann eine Rolle aus der Domäne einer web-basierten Applikation gelöscht werden.

- Der Dienst *showRoles* zeigt alle Rollen an, welche innerhalb der Domäne einer personalisierten, web-basierten Applikation zur Verfügung stehen.

- Der Dienst *configureRole* ermöglicht die Konfiguration einer Rolle, d.h. die Zuordnung von Features zu Rollen bzw. das Entfernen solcher Assoziationen.

- Der Dienst *classifyFeature* ermöglicht die Konfiguration einer Featureklasse, d.h. die Zuordnung von Features zu Featureklassen bzw. das Entfernen solcher Assoziationen.

11.3.2 Verwaltungs- und Navigationsfunktionalität

Personalisierte Applikationen müssen eine grundlegende Menge an Funktionalitäten zur Verfügung stellen, um

- Benutzer zu verwalten,

- die Rechte von Benutzern zu verwalten und
- die Benutzer entsprechend ihrer Rechte unterschiedlich und der Applikation angemessen behandeln zu können, z.B. im Rahmen der angebotenen Navigationsmöglichkeiten.

Diese Verwaltungs- und Navigationsfunktionalität (siehe Abschnitt 2.1) wird innerhalb des Personalisierungsframeworks in Form von Features vordefiniert und als eine Menge zugehöriger, wiederverwendbarer Basisdienste realisiert.

Abb. 11.16 illustriert anhand eines UML ([4, 5, 24]) Anwendungsfalldiagramms, welche Features dabei für welche Art von Benutzern im Rahmen einer personalisierten, web-basierten Applikation zur Verfügung stehen müssen.

Im Einzelnen wird dabei zwischen folgenden unterschiedlichen Typen von Benutzern unterschieden:

- Ein *nicht registrierter Benutzer* kann anonym die öffentlich zugänglichen Features einer Applikation nutzen, welche über eine öffentliche Navigation angeboten werden. Er kann zu einem registrierten Benutzer der Applikation werden, indem er das Feature 'registrieren' ausführt und auf diese Weise einen Benutzeraccount erhalten.
- Ein *registrierter Benutzer* kann sich mithilfe seines Benutzernamens und seines Passworts bei der Applikation anmelden und kann dann seine Benutzerdaten ändern. Ein registrierter Benutzer kann Zugang zu dem privaten Bereich der Applikation erlangen, indem er sich authentifiziert. Dabei werden seine Daten von einem Administrator auf ihre Korrektheit überprüft.
- Einem *authentifizierten Benutzer* werden die privaten Features der Applikation über eine mehrstufige, personalisierte Navigation angeboten. Der Aufbau dieser personalisierten Navigationsleiste orientiert sich an den Rollen, den zu den Rollen gehörigen Featureklassen und schließlich an den konkreten Features – wie in Abschnitt 11.4.3 dargestellt.
- Ausgezeichnete Benutzer haben z.B. als *Administrator* die Möglichkeit, die Verwaltung der Benutzeraccounts sowie der diesen zugeordneten Rollen zu übernehmen. Für diesen Zweck werden im Rahmen des Personalisierungsframeworks mehrere Dienste realisiert, wie in Abb. 11.16 dargestellt.

Diese über das Personalisierungsframework zur Verfügung gestellten Verwaltungs- und Navigationsfunktionalitäten bilden einen festen Bestandteil einer personlisierten, web-basierten Applikation. Bei deren Entwicklung werden diese Funktionalitäten durch Wiederverwendung von innerhalb des Frameworks vordefinierten und bereits realisierten Basisdiensten als Features in die Applikation eingebunden.

Ein Teil dieser Features legt die Navigationsfunktionalität innerhalb der Applikation fest:

- Das Feature 'öffentliche Navigation' (Dienst *showPublicNavbar*) dient der Visualisierung der öffentlichen Navigationsleiste, d.h. dem Bereitstellen der Einstiegspunkte in die verschiedenen öffentlichen Features bzw. der sie realisierenden Dienste der Applikation. Alle öffentlichen Dienste, sowie der Dienst *login* zum Anmelden bei der Applikation sind mit der öffentlichen Navigationsleiste gekoppelt.

- Das Feature 'personalisierte Navigation' (Dienst *showPrivateNavbar*) dient der Visualisierung der personalisierten Navigationsleiste, die einem Benutzer angezeigt wird, nachdem er sich bei Applikation via Benutzername und Passwort angemeldet hat. Alle privaten Features bzw. die sie realisierenden Dienste, die den Benutzern nur nach erfolgreicher Anmeldung bei der Applikation zur Verfügung stehen, sind mit der privaten Navigationsleiste gekoppelt (siehe Abschnitt 11.4.3).

- Das Feature 'anmelden' (Dienst *login*) ermöglicht es einem Benutzer, sich bei der Applikation anzumelden. Nach erfolgreicher Anmeldung gelangt der Benutzer in den privaten Bereich der Applikation.

- Das Feature 'personalisierte Begrüßung' (Dienst *showPrivateWelcome*) realisiert die interne Begrüßungsseite, die dem Benutzer nach der erfolgreichen Anmeldung bei der Applikation angezeigt wird.

- Das Feature 'Rollen-Homepage' (Dienst *showRoleHome*) realisiert die Rollen-Homepage, welche dem Benutzer nach der Auswahl einer Rolle aus der personalisierten Navigationsleiste angezeigt wird.

- Das Feature 'Funktionalitäts-Übersicht' (Dienst *showFeatureClassOptions*) realisiert die Übersicht über die einem Benutzer zur Auswahl stehenden Features, nachdem er eine Rolle und eine Featureklasse aus der personalisierten Navigationsleiste ausgewählt hat.

- Das Feature 'abmelden' (Dienst *logout*) ermöglicht es einem Benutzer, sich bei der Applikation abzumelden. Nach erfolgreicher Abmeldung gelangt der Benutzer zurück in den öffentlichen Bereich der Applikation.

Weitere Features ermöglichen die Verwaltung von Benutzern sowie deren Rechten:

- Das Feature 'registrieren' (Dienst *registerAsUser*) dient einem anonymen Benutzer dazu, sich bei der Applikation zu registrieren und damit eine Benutzerkennung zu erhalten.

- Über das Feature 'Profil ändern' (Dienst *changeUserProfileOwn*) kann ein registrierter Benutzer seine Benutzerdaten ändern.
- Feature 'Benutzeraccount anlegen' (Dienst *createUser*) dient dem Erzeugen und Einfügen eines neuen Benutzeraccounts in die Persistenzschicht der Applikation.
- Feature 'Benutzeraccount löschen' (Dienst *deleteUser*) dient dem Löschen eines Benutzeraccounts aus der Persistenzschicht der Applikation.
- Die Features 'Benutzer authentifizieren', 'Benutzer sperren' und 'Benutzer ablehnen' (Dienste *authenticateUser*, *lockUser* und *revokeUser*) werden zur Änderung des Status eines Benutzeraccounts innerhalb der Applikation verwendet.
- Feature 'Benutzerliste anzeigen' (Dienst *showUserListAll*) ermöglicht das Anzeigen einer Liste aller bei der Applikation registrierten Benutzer.
- Feature 'Benutzerdaten ändern' (Dienst *changeProfileOther*) erlaubt es dem Administrator einer Applikation, die Benutzerdaten beliebiger Benutzer zu verändern.
- Feature 'Rolle zuweisen' (Dienst *assignRole*) dient dem Zuweisen einer Rolle an einen Benutzer. Dieser Dienst ist an die jeweilige Applikation anzupassen, wenn applikations-spezifische Rollendaten im Datenmodell der Applikation vorgesehen sind und verwendet werden.
- Feature 'Rolle entziehen' (Dienst *revokeRole*) ermöglicht es, einem Benutzer eine zugewiesene Rolle wieder zu entziehen.
- Feature 'Rollendaten ändern' (Dienst *modifyRoleData*) erlaubt es dem Administrator einer Applikation, die Rollendaten beliebiger Benutzer zu verändern. Dieser Dienst ist nur dann nötig und muss entsprechend an die Applikation angepasst werden, wenn applikations-spezifische Rollendaten im Datenmodell der Applikation vorgesehen sind und verwendet werden.

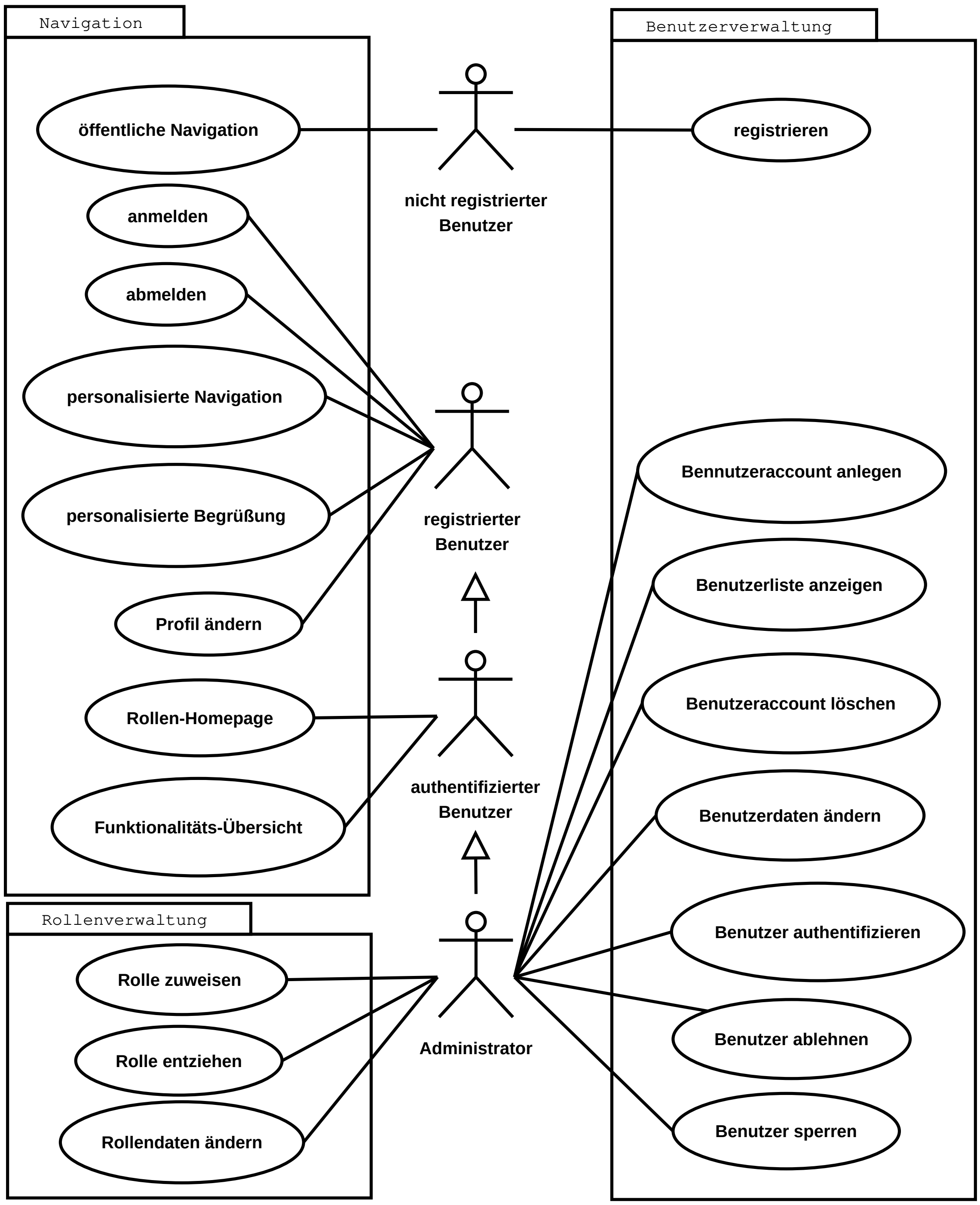


Abbildung 11.16: Verwaltungs- und Navigationsfunktionalität

11.4 Einsatzgebiete und Anwendungsbeispiele

Die in Abschnitt 11.1 eingeführten grundlegenden Begriffe der Personalisierung gehen an vielen Stellen während der Entwicklung und des Betriebs einer personalsierten, web-basierten Applikation ein. Im Folgenden sind einige Anwendungsbeispiele aus dem TEMPLUS Projekt zusammengestellt.

11.4.1 Konfiguration des Personalisierungsframeworks

Um das Personalisierungsframework mithilfe der zur Verfügung gestellten Adminsitrationsfunktionalität für eine Applikation zu konfigurieren, ist es notwendig, die in Abschnitt 11.1 eingeführten Begriffe für die Applikation festzulegen, d.h. die Domänenmodellierung für die Applikation vorzunehmen.

Diese Konfiguration bezieht sich auf die in Abb. 11.17 gekennzeichneten Begriffe (siehe Abschnitt 11.1), die im Rahmen der Entwicklung des Personalisierungsframeworks eingeführt wurden und vor dem Bereitstellen einer Applikation festgelegt werden müssen – wie in Abschnitt 11.3.1 beschrieben.

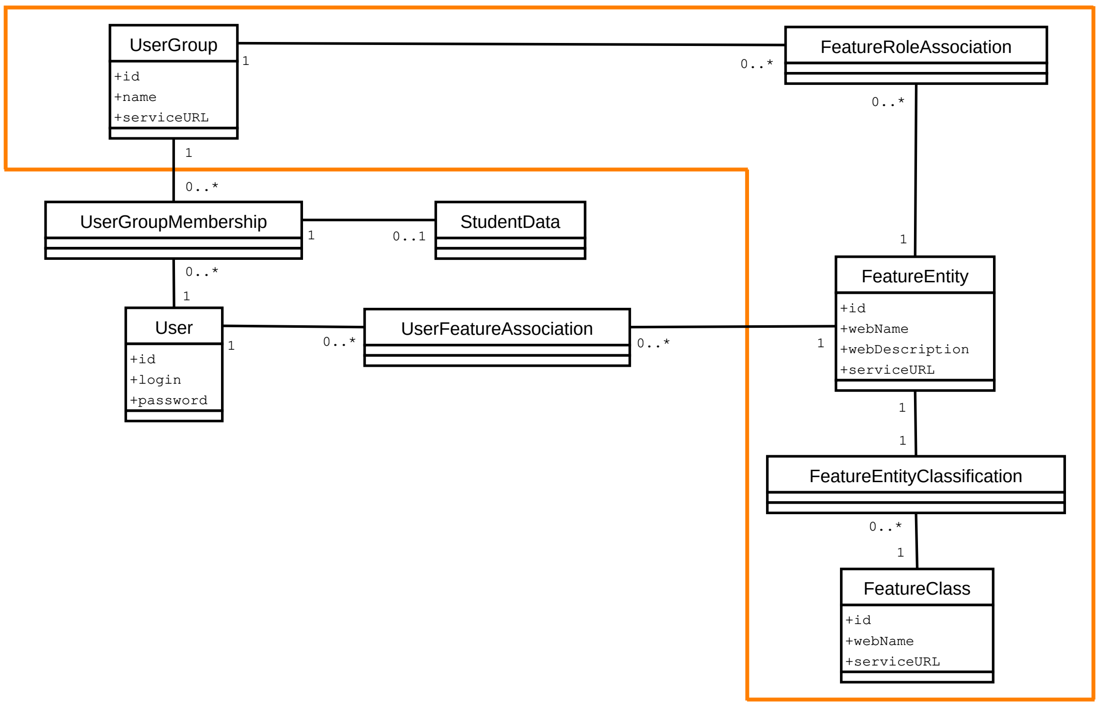


Abbildung 11.17: Konfiguration des Personalisierungsframeworks für TEMPLUS

Im Einzelnen umfasst dies folgende Aktivitäten:

- Anlegen, Löschen, Modifizieren der Features
- Anlegen, Löschen, Modifizieren der Featureklassen
- Zuordnung von Features zu Featureklassen sowie Löschen dieser Assoziationen
- Anlegen, Löschen, Modifizieren der Rollen
- Zuordnung von Features zu Rollen sowie Löschen dieser Assoziationen

Alle oben genannten Teilbereiche stehen in direktem Zusammenhang mit den Features der Applikation, d.h. der von der Applikation in Form von Diensten zur Verfügung gestellten Funktionalität.

Da dem Bereitstellen einer neuen Funktionalität die Implementierung derselben vorausgeht, entspricht das Bereitstellen einer neuen Funktionalität innerhalb der Applikation dem Bereitstellen der zugehörigen Implementierung. Dies ist aus technischen Gründen nur offline möglich, da dabei – aufgrund der notwendigen Änderungen in der Konfigurationsdatei `web.xml` – ein Neustart der Applikation erforderlich ist.

Das Definieren und Modifizieren von Rollen könnte zum Teil auch zur Laufzeit erfolgen, solange keine spezifischen Rollendaten gespeichert werden müssen und sich die Rollendefinition nur auf bereits innerhalb der Applikation verfügbare Features beschränkt. Darauf wurde jedoch im TEMPLUS Projekt verzichtet, da hier – wie in Abschnitt 11.2.5 beschrieben – Rollendaten verwendet werden.

11.4.2 Verwaltung von Benutzern und deren Rechten

Die Verwaltung von Benutzern und deren Rechten nimmt Bezug auf konkrete Benutzer (wie in Abb. 11.18 dargestellt) und kann daher nur zur Laufzeit einer Applikation durchgeführt werden.

Im Einzelnen umfasst dies folgende Aktivitäten – wie bereits in Abschnitt 11.3.2 beschrieben:

- Hinzufügen, Modifizieren und Löschen von Benutzeraccounts
- Zuweisung von Rollen an Benutzer, Entziehen von Rollen sowie ggf. Eingeben, Modifizieren bzw. Löschen von Rollendaten
- Zuweisung von einzelnen Features an Benutzer sowie Entziehen der Berechtigung, einzelne Features auszuführen

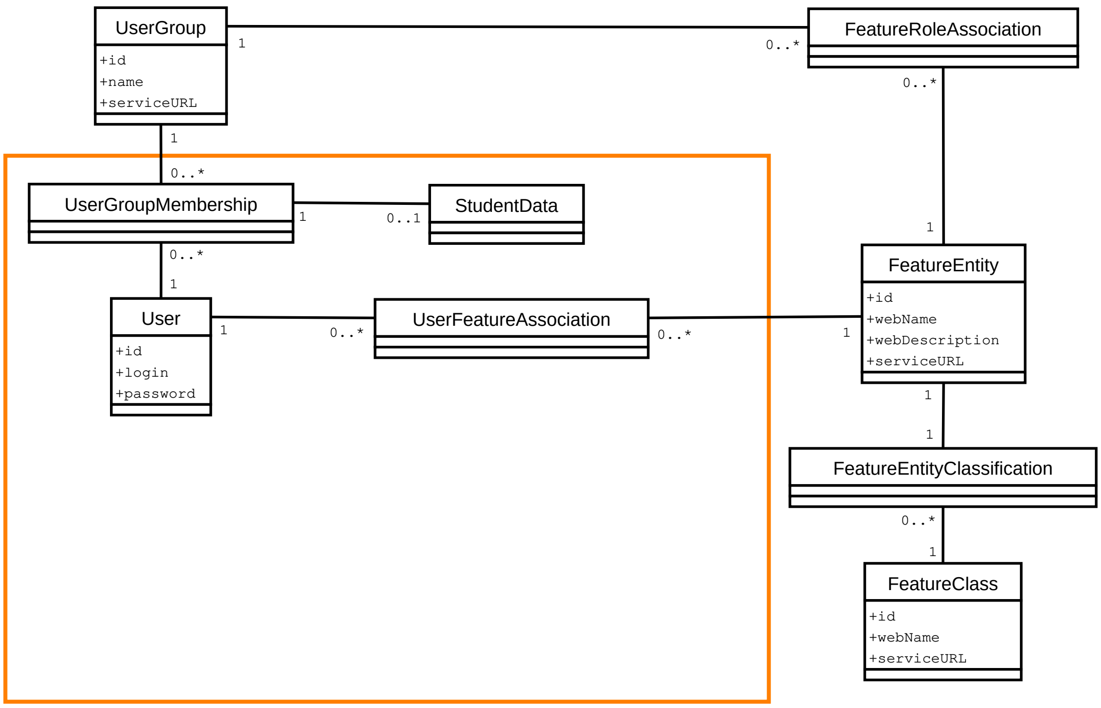


Abbildung 11.18: Verwaltung von Benutzern und deren Rechten in TEMPLUS

11.4.3 Navigation

Die Navigation ermöglicht einem Benutzer über eine Web-Oberfläche den Zugriff auf die Features einer Applikation und liefert eine strukturierte Anzeige der zur Verfügung stehenden Features.

Insbesondere bei komplexen web-basierten Applikationen, die sehr viele Features besitzen, spielt die strukturierte Anzeige eine wesentliche Rolle für die Benutzerfreundlichkeit der Applikation.

Wie bereits in Abschnitt 2.1 dargestellt, wird zwischen der öffentlichen und personalisierten Navigation unterschieden.

Personalisierte Navigation

Das TEMPLUS Projekt verwendet eine dreistufige personalisierte Navigation, welche die grundlegenden Begriffe der Personalisierung – Rolle, Featureklasse und Feature (siehe Abschnitt 11.1) – direkt abbildet.

- **Stufe 1: Rolle(n)**
  Nach dem Einloggen wird einem authentifizierten Benutzer seine persönliche

Navigationsleiste angezeigt, welche den Namen des Benutzers, sowie all seine Rollen beinhaltet (siehe Abb. 11.19). Der Benutzer kann dann eine Rolle auswählen bzw. zwischen seinen Rollen hin und her wechseln.

- **Stufe 2: Featureklasse(n)**
  Nachdem der Benutzer eine Rolle ausgewählt hat, werden ihm alle Featureklassen angezeigt, für welche ihm in der ausgewählten Rolle Features zur Verfügung stehen (siehe Abb. 11.20). Der Benutzer kann dann eine Featureklasse auswählen, zwischen den Featureklassen der ausgewählten Rolle hin und her wechseln oder eine andere Rolle wählen.

- **Stufe 3: Feature(s)**
  Nachdem der Benutzer zusätzlich zur Rolle auch eine Featureklasse ausgewählt hat, werden ihm alle Features angezeigt, welche ihm in der ausgewählten Rolle und der ausgewählten Featureklasse zur Verfügung stehen (siehe Abb. 11.21). Der Benutzer kann dann ein konkretes Feature auswählen, zwischen den angebotenen Features, welche zu der gewählten Rolle und gewählten Featureklasse gehören, hin und her wechseln, eine andere Featureklasse oder eine andere Rolle wählen.

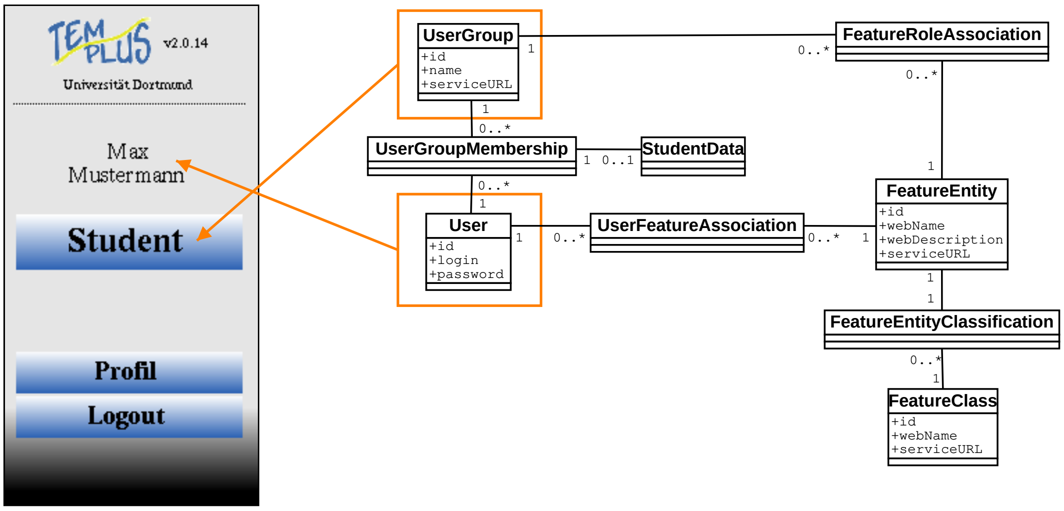


Abbildung 11.19: Personalisierte Navigation (1)

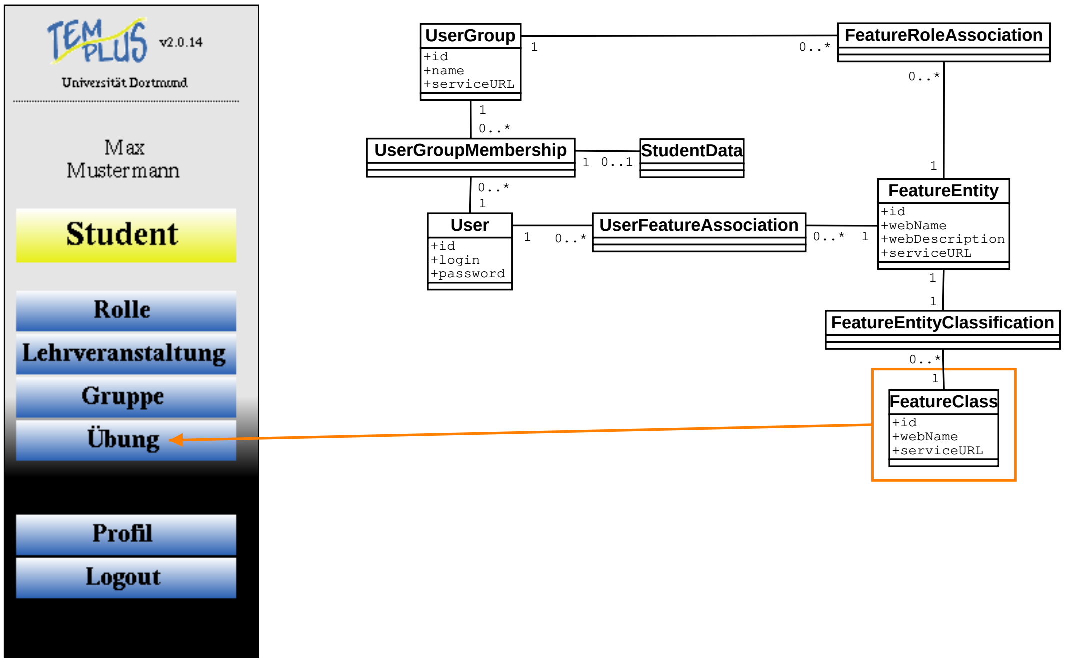


Abbildung 11.20: Personalisierte Navigation (2)

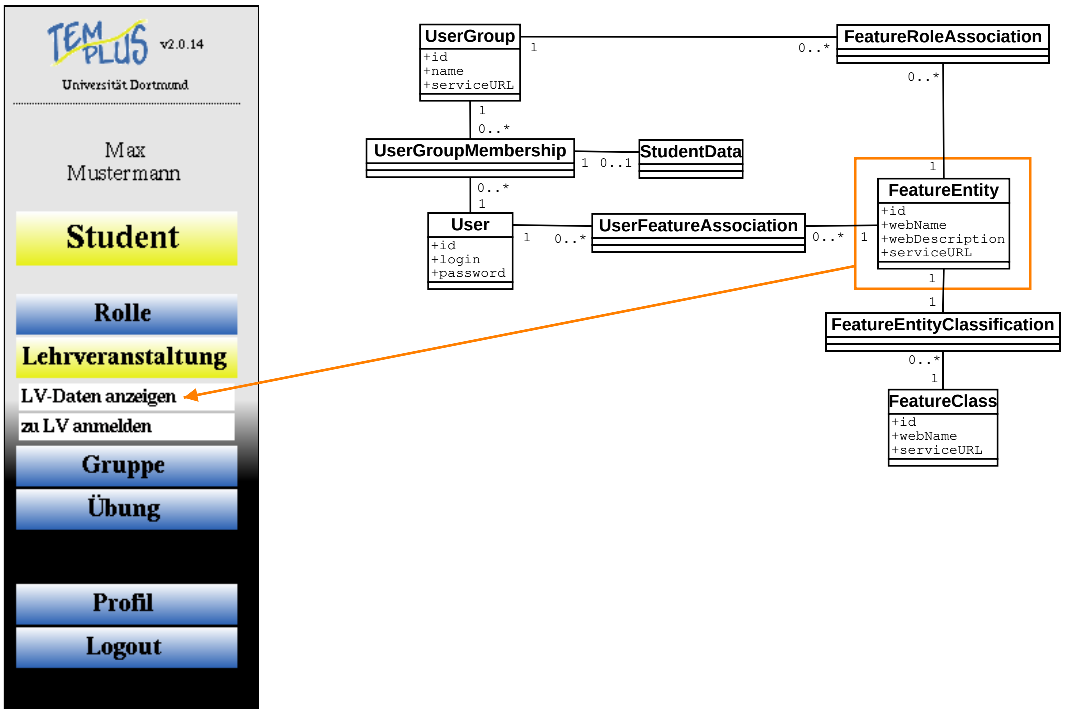


Abbildung 11.21: Personalisierte Navigation (3)

Öffentliche Navigation

Die öffentliche Navigation des TEMPLUS Projekts ist einstufig.

- **Stufe 1: Feature(s)**
  Alle öffentlich zugänglichen Features (d.h. externe Dienste vom Typ `PUBLIC` bzw. `LOGIN`) werden dem Benutzer in einer Auswahlliste angeboten.

Dies ist hier sinnvoll, da es im TEMPLUS Projekt insgesamt nur 6 Features dieser Art gibt ('Registrierung für Studierende', 'Registrierung für Mitarbeiter', 'Login', 'Datenschutzerklärung', 'Information', 'Neues Passwort anfordern').

Stellt eine web-basierte Applikation eine wesentlich größere Menge an öffentlich zugänglichen Features zur Verfügung, kann die öffentliche Navigation mithilfe der Featureklassen besser strukturiert werden, indem sie zweistufig angelegt wird:

- **Stufe 1: Featureklasse(n)**
  Einem Benutzer werden alle Featureklassen angezeigt, für die es öffentliche zugängliche Features gibt (d.h. externe Dienste vom Typ `PUBLIC` bzw. `LOGIN`).
- **Stufe 2: Feature(s)**
  Nach der Auswahl einer Featureklasse werden dem Benutzer alle öffentlich zugänglichen Features angezeigt, die der ausgewählten Featureklasse zugeordnet sind.

Dies kann durch eine alternative Realisierung des Features 'öffentliche Navigation' (Basisdienst *showNavbarPublic*), welches durch das Personalisierungsframework zur Verfügung gestellt wird, erreicht werden.

11.4.4 Zugriffsberechtigungsprüfung

Ein Benutzer interagiert mit einer web-basierten Applikation

- durch das direkte Ausführen von URLs (Eingabe in den Browser)

oder

- das indirekte Ausführen von URLs (Anklicken eines Links).

Hierbei ist prinzipiell auch Bookmarking möglich, d.h. ein Benutzer kann jeden Interaktionspunkt einer web-basierten Applikation in seinem Browser speichern und dann später direkt anwählen.

Bei jeder Interaktion mit einer Applikation bzw. einem konkreten Feature dieser muss demnach geprüft werden, ob der aktuelle Benutzer das Recht hat, das angesprochene Feature auszuführen.

Hierfür werden einerseits Informationen über den aktuellen Benutzer in dessen Session gehalten, z.B. ob der Benutzer sich bei der Applikation angemeldet hat und welche Rollen er besitzt. Andererseits werden mit der URL, die für die Interaktion mit dem Feature benutzt wird, als Aufrufparameter die eindeutigen Ids für die ausgewählte Rolle, die ausgewählte Featureklasse und das ausgewählte Feature mit übergeben.

Innerhalb der Applikation kann dann mithilfe von Basiskomponenten im Rahmen des Personalisierungsframeworks geprüft werden, ob

- der aktuelle Benutzer die angegebene Rolle innehat,
- ihm in dieser Rolle Features aus der angegebenen Featureklasse zur Verfügung stehen,
- er das Recht hat, das angesprochene Feature auszuführen, und
- die übergebenen Ids zu dem angesprochenen Feature passen.

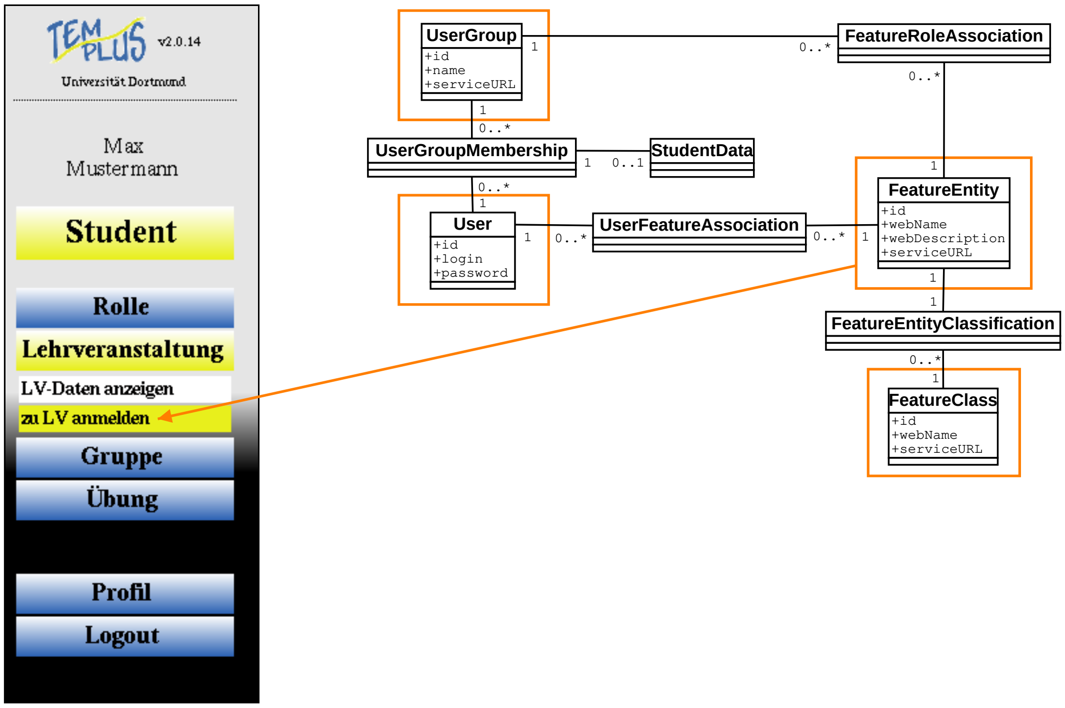


Abbildung 11.22: Zugriffsberechtigungsprüfung

11.4.5 Benutzer-spezifische Filterung von Daten

Im Rahmen von personalisierten, web-basierten Applikationen kann die Verfügbarkeit von Features nicht nur von den Rechten des jeweiligen Benutzers abhängen, sondern an zusätzliche Filtermechanismen im Rahmen der Berechtigungsprüfung gekoppelt werden.

Diese zusätzlichen Filtermechanismen prüfen dann den Bezug des aktuellen Benutzers zu einem speziellen Typ von Geschäftsobjekten aus dem Datenmodell der Applikation.

Einen Spezialfall für die benutzer-spezifische Filterung von Daten stellt die personalisierte Navigation dar, welche in Abschnitt 11.4.3 beschrieben ist.

Die benutzer-spezifische Filterung von Daten wird an all jenen Stellen notwendig bzw. möglich, wo im Datenmodell der Applikation Objekte vom Typ `User` in Relation stehen mit einem speziellen Typ von Geschäftsobjekten der Applikation und diese Assoziation innerhalb der Persistenzschicht gespeichert ist.

Das TEMPLUS Projekt bietet dafür verschiedene Beispiele. Im Lehremanagement an einerUniversität gibt es u.a. folgende Szenarien:

- Benutzer mit Rolle *Organisator* organisieren Lehrveranstaltungen.
  Ein Objekt vom Typ `User` ist dabei über eine Relation `isOrganizer` mit einem Objekt vom Typ `Course` assoziiert.
- Benutzer mit Rolle *Dozent* bieten Lehrveranstaltungen an.
  Ein Objekt vom Typ `User` ist dabei über eine Relation `isLecturer` mit einem Objekt vom Typ `Course` assoziiert.
- Benutzer mit Rolle *Tutor* moderieren Gruppen einer Lehrveranstaltung.
  Ein Objekt vom Typ `User` ist dabei über eine Relation `isTutor` mit einem Objekt vom Typ `Group` assoziiert.
- Benutzer der Rolle *Student* nehmen an Lehrveranstaltungen teil.
  Ein Objekt vom Typ `User` ist dabei über eine Relation `isParticipant` mit einem Objekt vom Typ `Course` assoziiert.

Diese Relationen können im Rahmen der Realisierung von Features für eine benutzer-spezifische Filterung der verwendeten Daten eingesetzt werden:

Beispiel 11.4:

Betrachten wir die Featureklasse 'Lehrveranstaltung'. Dabei soll jeweils sichergestellt sein, dass nur berechtigte Personen die Teilnehmerliste einer Lehrveranstaltung einsehen dürfen.

Folgende zusätzliche Filtermechanismen werden in diesem Beispiel angewendet, um die Menge der für das Feature *'Teilnehmerliste einer Lehrveranstaltung anzeigen'* zur Auswahl angebotenen Lehrveranstaltungen zu bestimmen.

Eine Lehrveranstaltung erscheint in der Auswahlliste eines Benutzers, wenn er die Featureklasse *Lehrveranstaltung* und das Feature *'Teilnehmerliste einer Lehrveranstaltung anzeigen'* ausgewählt hat und eine der folgenden Bedingungen gilt:

- Der aktuelle Benutzer hat die Rolle *Organisator* ausgewählt und ist als Organisator der Lehrveranstaltung zugewiesen.
- Der aktuelle Benutzer hat die Rolle *Dozent* ausgewählt und ist als Dozent der Lehrveranstaltung zugewiesen.

Das bedeutet also, dass Benutzer nur Zugriff auf die Teilnehmerlisten von Lehrveranstaltungen haben, die in ihrem Zuständigkeitsbereich liegen. ⌟

Beispiel 11.5:

Betrachten wir nun die Featureklasse 'Gruppe'. Dabei soll jeweils sichergestellt sein, dass nur berechtigte Personen die Teilnehmerliste einer Übungsgruppe einer Lehrveranstaltung einsehen dürfen.

Folgende zusätzliche Filtermechanismen werden in diesem Beispiel angewendet, um die Menge der für das Feature *'Teilnehmerliste einer Gruppe einer Lehrveranstaltung anzeigen'* zur Auswahl angebotenen Lehrveranstaltungen zu bestimmen.

Hat ein Benutzer die Featureklasse *Gruppe* und das Feature *'Teilnehmerliste einer Gruppe einer Lehrveranstaltung anzeigen'* ausgewählt, dann gilt:

- Hat der aktuelle Benutzer die Rolle *Organisator* ausgewählt und ist als Organisator der Lehrveranstaltung zugewiesen, so erhält er Zugriff auf die Teilnehmerlisten aller Übungsgruppen dieser Lehrveranstaltung.
- Hat der aktuelle Benutzer die Rolle *Tutor* ausgewählt und ist als Tutor mindestens einer Gruppe der Lehrveranstaltung zugewiesen, so erhält er Zugriff auf die Teilnehmerlisten der Übungsgruppen dieser Lehrveranstaltung, für die er als Tutor eingetragen ist.
- Hat der aktuelle Benutzer die Rolle *Student* ausgewählt und ist als Teilnehmer bei der Lehrveranstaltung angemeldet, so erhält er Zugriff auf die Information, für welche Übungsgruppe dieser Lehrveranstaltung er eingeteilt ist.

⌟

Kapitel 12

Leitfaden zum Einsatz des Personalisierungsframeworks

Personalisierte, web-basierte Applikationen müssen sich mit der Verwaltung von Benutzern, Benutzergruppen und ihren Rechten auseinandersetzen und entsprechende Features für diese Anforderungen definieren sowie realisieren. Der Einsatz des in Kapitel 11 eingeführten Personalisierungsframeworks deckt diese Anforderungen an die Menge der Features einer Applikation durch die Wiederverwendung der dafür notwendigen vordefinierten Basisdienste ab, welche bereits in Abschnitt 11.3.2 detailliert beschrieben worden sind. Auf diese Weise wird der Arbeitsaufwand für die Realisierung der Applikation erheblich reduziert.

Einige Adaptions- sowie Konfigurationsaufgaben müssen aber dennoch durchgeführt werden, damit das entwickelte Personalisierungsframework für eine neue personalisierte, web-basierte Applikation zum Einsatz kommen und korrekt und zuverlässig funktionieren kann.

Folgende Vorbereitungen und Arbeitsschritte sind dafür notwendig:

1. Erstellen der Anforderungsdefinition
2. Auswahl der wiederverwendbaren Dienste
3. Aufsetzen der Konfigurationsapplikation
4. Durchführen der Domänenmodellierung
5. Adaption der wiederverwendeten Basisdienste
6. Realisierung der applikations-spezifischen Features

Die folgenden Abschnitte beschreiben die Vorgehensweise sowie die wichtigsten Arbeitsaufgaben beim Entwickeln einer neuen Applikation unter Verwendung des in Kapitel 11 beschriebenden Personalisierungsframeworks und gehen kurz auf notwendige Adaptions- sowie Konfigurationsschritte ein.

12.1 Anforderungsdefinition

Die Anforderungsdefinition für die neue Applikation erfolgt i.d.R. mittels UML. Dabei wird ein Anwendungsfalldiagramm erstellt, welches die grundlegenden Begriffe der Personalisierung – Rollen, Featureklassen und Features sowie deren Zuordnung zueinander – festlegt, wie bereits in Abschnitt 11.1 beschrieben.

Dabei wird die Gesamtfunktionalität der zu entwickelnden Applikation in Features aufgeteilt und eine Klassifizierung dieser zu Featureklassen vorgenommen. Darüber hinaus werden die Rollen der Benutzer spezifiziert, die die Applikation unterstützen soll, und die Rechte dieser Rollen festgelegt.

Das Anwendungsfalldiagramm für eine neue personalisierte, web-basierte Applikation kann als eine Erweiterung des in Abb. 11.16 dargestellten Diagramms entwickelt werden, welches bereits die Navigations- und Verwaltungsfunktionalität des Personalisierungsframeworks beinhaltet.

12.2 Wiederverwendung

Die im Rahmen der Anforderungsdefinition (siehe Abschnitt 12.1) identifizierten Features werden in drei disjunkte Mengen aufgeteilt:

- Features, welche mithilfe von vordefinierten, wiederverwendbaren Diensten aus dem Personalisierungsframework realisiert werden können,
- Features, für welche vordefinierte, wiederverwendbare Dienste aus dem Personalisierungsframework als Vorlage dienen und eine Wiederverwendung nur nach einer entsprechenden, geringfügigen Erweiterung bzw. Anpassung dieser Dienste möglich ist, und
- applikations-spezifische Features, welche im Rahmen der Entwicklung der Applikation von Grund auf umgesetzt werden müssen.

Für die erste Menge von Features kann dann eine Auswahl der Dienste aus dem Personalisierungsframework getroffen werden, welche im Rahmen der Applikation wiederverwendet werden, um eben diese Menge von Features zu realisieren.

Auf die zweite Menge von Features geht Abschnitt 12.5 näher ein.

Die Menge der applikations-spezifischen Features wird in Abschnitt 12.6 genauer betrachtet.

12.3 Konfigurationsapplikation

Um das Personalisierungsframework für eine neue personalisierte, web-basierte Applikation einsetzen zu können, muss zunächst die Konfigurationsapplikation, welche die Administrationsfunktionalität des Personalisierungsframeworks in Form einer eigenständigen web-basierten Applikation enthält, zur Benutzung bereitgestellt werden.

Beim Aufsetzen der Konfigurationsapplikation wird ferner eine Datenbank mit den Tabellen für die grundlegenden Begriffe der Personalisierung angelegt. In diese werden alle Informationen für die Navigations- und Verwaltungsfunktionalität des Personalisierungsframeworks eingetragen. Damit stehen die in Abb. 11.16 dargestellten Rollen, Featureklassen, Features und Rechte bereits zur Verfügung.

12.4 Domänenmodellierung

Die in Abschnitt 12.1 durch ein UML Anwendungsfalldiagramm festgelegten Begriffe für den Bereich der Personalisierung, wie sie im Rahmen dieser Arbeit verwendet wird, können dann vom Personalisierungsexperten mithilfe der Konfigurationsapplikation (siehe Abschnitt 12.3) in die Persistenzschicht der neuen Applikation übertragen werden.

Auf diese Weise wird die Domänenmodellierung für diese Applikation vorgenommen, d.h. das Personalisierungsframework wird administriert bzw. konfiguriert.

Für die Administration des Personalisierungsframeworks werden folgende Basisdienste genutzt, um die innerhalb des Anwendungsfalldiagramms definierten Begriffe für die zu entwickelnde, personalisierte, web-basierte Applikation festzulegen:

- Anlegen der Rollen: Für jeden Akteur aus dem Anwendungsfalldiagramm, welcher sich nicht auf den Status eines Benutzers ('nicht registriert', 'registriert', 'authentifiziert') bezieht und somit einen applikations-spezifischen Akteur darstellt, wird eine Rolle gleichen Namens angelegt.

- Anlegen der Featureklassen: Für jedes Paket aus dem Anwendungsfalldiagramm wird eine Featureklasse gleichen Namens angelegt.

- Anlegen der Features: Für jeden Anwendungsfall aus dem Anwendungsfalldiagramm wird ein Feature gleichen Namens angelegt. Dabei müssen für jedes Feature noch zusätzliche Eigenschaften festgelegt werden:
  - Webname
  - Kurzbeschreibung
  - URL des Dienstes, welcher den Anwendungsfall realisiert
  - Sichtbarkeit: externe Dienste, welche in der Navigationsleiste erscheinen sollen, werden als 'sichtbar' definiert, alle anderen als 'nicht sichtbar'
  - Typ: 'öffentlich', wenn der Anwendungsfall von der Rolle *nicht registrierter Benutzer* ausgeführt werden darf, 'privat', wenn der Anwendungsfall von der Rolle *registrierter Benutzer* oder einer daraus abgeleiteten Rolle ausgeführt werden darf.
- Die Zuordnung von Features zu Featureklassen erfolgt gemäß der Zuordnung der entsprechenden Anwendungsfälle zu den Paketen im Anwendungsfalldiagramm.
- Zuordnung von Features zu Rollen erfolgt gemäß der Berechtigung, welche durch die Verbindungen zwischen Akteuren und Anwendungsfällen aus dem Anwendungsfalldiagramm abzulesen sind.

12.5 Basisdienste

Die zweite Menge der in Abschnitt 12.2 identifizierten Dienste, welche nur nach geringfügiger Anpassung wiederverwendet werden können, ist Gegenstand dieses Arbeitsschrittes.

Dabei wird an einigen Stellen eine applikations-spezifische Erweiterung bzw. Anpassung des Personalisierungsframeworks vorgenommen, d.h. seiner Verwaltungs- und Navigationsfunktionalität.

Diese Erweiterung und Anpassung muss mithilfe des *ABC-SD* erfolgen und umfasst folgende Bereiche:

- Der OO-Spezialist nimmt eine ***programmatische Erweiterung*** durch die Realisierung neuer Rollendaten vor, die im Rahmen der Applikation zum Einsatz kommen sollen.
- Der Anwendungsexperte führt für die entsprechenden Dienste eine ***Anpassung und Erweiterung der SLGs*** durch, z.B. des Dienstes *addRole*, wenn neue Rollendaten im Rahmen der Applikation berücksichtigt werden müssen.

- Der HTML- bzw. GUI-Designer führt die ***Anpassung der GUI***, d.h. der HTML Seiten, der entsprechenden Dienste an die Bedürfnisse der Applikation (z.B. Layout) bzw. an die innerhalb der *SLG* vorgenommenen Änderungen durch.

12.6 Applikations-spezifische Features

Die Realisierung der applikations-spezifischen Features kann – muss aber nicht – mit dem *ABC-SD* erfolgen.

Das Personalisierungsframework ist auch für eine personalisierte, web-basierte Applikation einsetzbar, deren applikations-spezifische Features nicht mit dem *ABC-SD* realisiert werden, da ja im Rahmen der Domänenmodellierung für eine Applikation (siehe Abschnitt 12.4) lediglich die URL – d.h. die Einsprungstelle – für jedes einzelne Feature angegeben wird.

Dennoch bietet eine Realisierung der kompletten Applikation, d.h. auch der applikations-spezifischen Features, einige Vorteile.

Das folgende Kapitel 13 beleuchtet den gesamten Entwicklungsprozess einer personalisierten, web-basierten und als Dienstfamilie konzipierten Applikation unter Verwendung des *ABC-SD*.

Teil IV

Framework – Prozess

Kapitel 13

Entwicklungsprozess

Im Rahmen der praktischen Anwendung des in Kapitel 8 beschriebenen Prozessmodells $PM_{ABC\text{-}SD}$ bei der Entwicklung von zuverlässigen, personalisierten, web-basierten Applikationen mit dem *ABC-SD* muss zunächst eine Anpassung dieses Prozessmodells an die projekt-spezifischen Randbedingungen erfolgen, welche sich beispielsweise durch die Applikationsstruktur oder den Einsatz bestimmter vorgefertigter Bibliotheken ergeben:

- Die komplexe, personalisierte, web-basierte Applikation TEMPLUS ist als Dienstfamilie konzipiert – wie in Abschnitt 2.1 dargestellt.
- Die Applikation wird unter Verwendung der vorgefertigten Bibliothek *EWIS* User, welche ein Persistenzframework für eine web-basierte Applikation bereitstellt, mithilfe des *ABC-SD* realisiert.
- Die Domänenmodellierung für die Applikation erfolgt durch Einsatz des in vorgestellten Personalisierungsframeworks – wie in Kapitel 12 beschrieben.

Abb. 13.1 zeigt den im Rahmen des TEMPLUS Projektes verwendeten Software-Entwicklungsprozess $MyProcess_{ABC\text{-}SD}$[1] in Form eines UML Aktivitätsdiagramms.

[1]Die Abbildung 13.1 enthält ausschließlich englische Texte, da es sich hier um eine an die projekt-spezifischen Anforderungen von TEMPLUS angepasste und erweiterte Darstellung des Prozessmodells aus der in Englisch verfassten Dissertation [11] handelt. Um den Bezug zu der originalen Abbildung des dort eingeführten Prozessmodells (siehe auch Abb. 8.1) herzustellen und um Irritationen zu vermeiden, wurde daher auf eine Übersetzung verzichtet sowie die Erweiterung und Verfeinerung ebenfalls unter Verwendung englischer Texte vorgenommen.

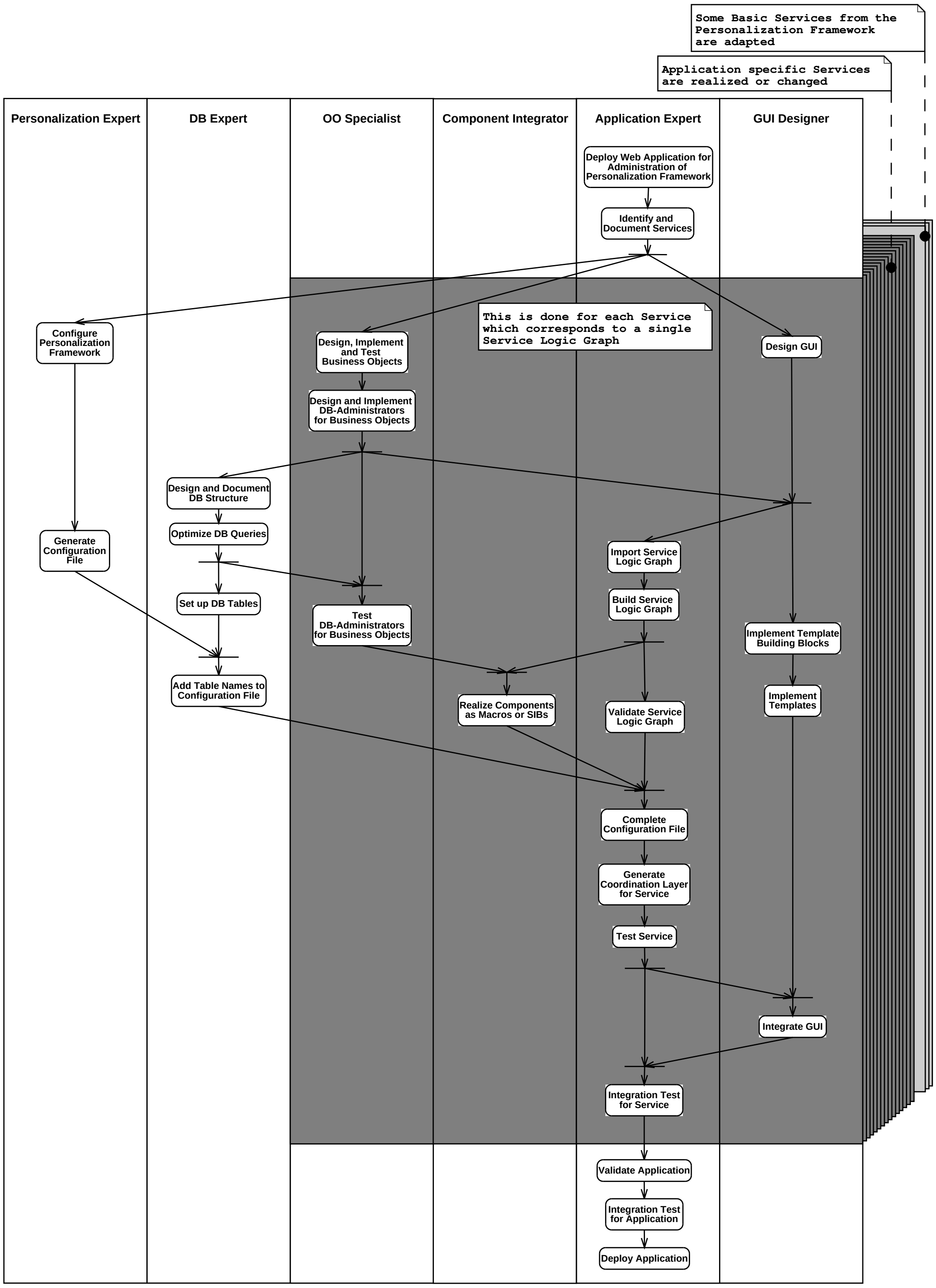


Abbildung 13.1: Der Software-Entwicklungsprozess $MyProcess_{ABC\text{-}SD}$

Dabei ist zu beachten, dass Abb. 13.1 eine idealisierte Darstellung des Entwicklungsprozesses widerspiegelt. Wie in jedem inkrementellen Entwicklungsprozess können sowohl die Analyse- und Modellierungsphase, als auch die Aktivitäten, welche in der Abbildung präsentiert werden, wiederholt ausgeführt werden, d.h. jedes Mal bei einer Erweiterung oder Änderung der zu entwickelnden Applikation.

Die folgenden Abschnitte gehen näher auf die in Abb. 13.1 dargestellten Abläufe ein.

- Abschnitt 13.1 beschreibt die Aufgabenbereiche der verschiedenen am Entwicklungsprozess beteiligten Personen. Dabei werden die aufgrund projektspezifischer Randbedingungen notwendigen Erweiterungen diskutiert. Durch das Einführen von zwei neuen Rollen und durch die Erweiterung des Aufgabenbereichs bestehender Rollen wird die Entwicklungsarbeit stärker strukturiert und besser an die neue Struktur einer komplexen, personalisierten, web-basierten und als Dienstfamilie konzipierten Applikation angepasst.

- Die in Abschnitt 13.2 enthaltene klare Definition, Klassifizierung und Beschreibung der bestehenden Abhängigkeiten innerhalb des Entwicklungsprozesses $MyProcess_{ABC\text{-}SD}$ ermöglicht eine bessere Koordination und Kontrolle der Entwicklungsarbeit.
  In Teil V werden schließlich Lösungswege aufgezeigt, wie die durch die arbeitsteilige Software-Entwicklung bedingten Probleme bestmöglich in den Griff zu bekommen sind. Die Abhängigkeiten können dabei sowohl im Rahmen von automatischen Generierungsprozessen die Kommunikation zwischen den unterschiedlichen, am Prozess beteiligten Personen ergänzen, als auch bei der Realisierung entsprechender Konsistenzüberprüfungen berücksichtigt werden.

13.1 Rollen der am Prozess beteiligten Personen

Die Aufgaben während der Entwicklung personalisierter, web-basierter Applikationen mit *ABC-SD* werden – wie durch das Prozessmodell $PM_{ABC\text{-}SD}$ (siehe Abb. 8.1, [11]) vorgegeben – den entsprechenden Rollen der am Entwicklungsprozess beteiligten Personen zugeordnet.

Eine verfeinerte Darstellung (in Abb. 13.1 grau hinterlegt) des Software-Entwicklungsprozesses $PM_{ABC\text{-}SD}$, welche an die projekt-spezifischen Rahmenbedingungen angepasst ist, beschreibt hierbei das Vorgehen bei der Entwicklung eines einzelnen Dienstes im Rahmen einer als Dienstfamilie konzipierten, web-basierten Applikation.

Dieses Vorgehensmodell wird dann eingebettet in einen Prozessrahmen, welcher zusätzliche Aktivitäten für die Entwicklung einer als Dienstfamilie konzipierten, personalisierten, web-basierten Applikation enthält – wie in Abb. 13.1 dargestellt.

- Der Aufgabenbereich des Anwendungsexperten wird um Aktivitäten erweitert, welche sich aus der Entwicklung einer web-basierten Applikation als Dienstfamilie (siehe Abschnitt 2.1) ergeben, wichtig sind hier insbesondere die Zerlegung der Gesamtfunktionalität in einzelne Features und das Zusammenführen der Ergebnisse am Ende.

Andere Aufgabenbereiche werden aufgrund der zu berücksichtigenden prozess-spezifischen Randbedingungen explizit im Entwicklungsprozess aufgeführt und durch zwei neue Rollen beschrieben:

- Der Personalisierungsexperte übernimmt Aufgaben, welche sich aus dem Einsatz des in Kapitel 11 vorgestellten Personalisierungsframeworks ergeben.

- Der DB-Experte ist verantwortlich für die Tätigkeiten, die in direktem Zusammenhang stehen mit dem im Rahmen des *ABC-SD* verwendeten Persistenzframework ([48]).

Bei der Verfeinerung des Vorgehensmodells werden darüber hinaus bestimmten Rollen zusätzliche Aufgaben zugeordnet.

Auch bisher in $PM_{ABC\text{-}SD}$ nicht explizit aufgeführte Aktivitäten (z.B. Validierung) finden sich somit in dem Software-Entwicklungsprozess $MyProcess_{ABC\text{-}SD}$ wieder, wenn sie für die Spezifikation der Schnittstellen zwischen den verschiedenen Rollen oder die Verdeutlichung des Ablaufes notwendig sind.

Entsprechend des erweiterten Aufgabenbereichs wird an zwei Stellen eine Umbenennung der Rollen vorgenommen, da bisher verwendete Rollennamen die neuen Aufgaben nicht deutlich genug referenzieren:

- Der Aufgabenbereich der ursprünglichen Rolle des *SIB*-Integrators wird um Arbeitsaufgaben erweitert, die neben der Implementierung von *SIB*s auch die Konfiguration von Makros beinhalten. Es ergibt sich also eine Umbenennung der ursprünglichen Rolle *SIB-Integrator* zur Rolle *Komponenten-Integrator*, entsprechend der Erweiterung ihres Aufgabenbereiches.

- Der Aufgabenbereich der ursprünglichen Rolle des HTML-Designers beinhaltet das Design sowie die Realisierung der Benutzeroberfläche der web-basierten Applikation. Im TEMPLUS Projekt besteht das Oberflächendesign einer Applikation neben der Erstellung von HTML-Seiten auch in der Entwicklung anderer Dokumentenformate im Rahmen der Interaktion mit den Benutzern, z.B. Dateien der Formate CSV[2], XML[3], PDF[4] oder ASCII-Text (Endung `.txt`). Daher wird diese Rolle zur Rolle *GUI-Designer* umbenannt.

Die folgenden Abschnitte beschreiben die verschiedenen Rollen der am Entwicklungsprozess beteiligten Personen. So werden die einzelnen Tätigkeiten identifiziert und detailliert erklärt, welche innerhalb des in Abb. 13.1 dargestellten, verfeinerten und erweiterten Software-Entwicklungsprozesses ausgeführt werden müssen. Dieser kommt bei der Realisierung einer als Dienstfamilie konzipierten, personalisierten, web-basierten Applikation zum Einsatz.

Die Beschreibung der einzelnen Rollen geht dabei jeweils kurz auf die durch das Vorgehensmodell vorgegebenen Abhängigkeiten ein. In Abschnitt 13.2 sind diese dann detaillierter dargestellt.

13.1.1 System-Ingenieur

Der Aufgabenbereich des System-Ingenieurs bleibt im Vergleich zu dem Software-Entwicklungsprozess $PM_{ABC\text{-}SD}$ aus Kapitel 8 unverändert.

Seine Aktivitäten sind hauptsächlich auf die Analyse- und Modellierungsphase beschränkt, daher wird diese Rolle – wie auch in Kapitel 8– nicht in der Abbildung aufgeführt.

Der System-Ingenieur analysiert demnach das Problem, das mithilfe einer personalisierten, web-basierten Applikation gelöst werden soll. Er hält den Kontakt mit dem Kunden und zeichnet verantwortlich für das externe Verhalten der zu entwicklenden Applikation.

Außerdem ist der System-Ingenieur auch zuständig für die Umgebung, in welche die Applikation integriert werden soll, und für die Erfüllung von nicht-funktionalen Anforderungen, wie z.B. Performanz und Sicherheit.

Im Einzelnen entstehen hier – wie in [11] detailliert beschrieben – folgende Artefakte:

- Das initiale Anwendungsfalldiagramm, welches zunächst ohne Berücksichtigung des Personalisierungsframeworks die Gesamtfunktionalität der zu entwickelnden Applikation modelliert, wird erstellt und an den Anwendungsexperten (siehe Abschnitt 13.1.2) weitergegeben.

[2] Character Separated Values. Dateien mit Endung `.csv`
[3] Extensible Markup Language. Dateien mit Endung `.xml`
[4] Portable Document Format. Dateien mit Endung `.pdf`

- Die Beschreibung der Geschäftsprozesse wird spezifiziert und ebenfalls an den Anwendungsexperten übermittelt.
- Die Geschäftsklassen sowie ihre Verantwortlichkeiten und ihre Zusammenarbeit werden auf konzeptioneller Ebene beschrieben. Dies bildet die Grundlage für die Aufgaben, welche der OO-Spezialist (siehe Abschnitt 13.1.4) zu erfüllen hat.
- Das Zustandsübergangsdiagramm für die GUI-Modellierung wird mit dem Kunden abgestimmt und stellt die Ausgangsbasis für die Aufgaben des GUI-Designers (siehe Abschnitt 13.1.7) dar.

13.1.2 Anwendungsexperte

Der Aufgabenbereich des Anwendungsexperten ist zentral innerhalb des Entwicklungsprozesses $MyProcess_{ABC\text{-}SD}$.

Der Anwendungsexperte arbeitet mit allen anderen Rollen direkt oder indirekt zusammen. In seinem Aufgabenbereich werden die von den anderen Rollen gelieferten Ergebnisse schließlich vereint, bevor die fertige Applikation bereit gestellt werden kann.

Der Entwicklungsprozess $MyProcess_{ABC\text{-}SD}$ definiert das Vorgehensmodell für die Entwicklung einer personalisierten, web-basierten Applikation unter Verwendung des in Kapitel 11 eingeführten Personalisierungsframeworks. In einem ersten Schritt wird daher vom Applikationsexperten die web-basierte Applikation bereitgestellt, welche die Administrationsfunktionalität für das Personalisierungsframework (siehe Abschnitt 11.3.1) zur Verfügung stellt und damit die Domänenmodellierung (siehe Abschnitt 12.4) für die Applikation erst ermöglicht.

Der Anwendungsexperte steht außerdem in engem Kontakt zum System-Ingenieur (siehe Abschnitt 13.1.1) und dem Kunden und erhält von ihnen die Beschreibung und Dokumentation der Geschäftsprozesse sowie das initiale Anwendungsfalldiagramm.

Im nächsten Schritt wird dann die durch diese Geschäftsprozesse beschriebene, erwartete Gesamtfunktionalität der Applikation, d.h. das initiale Anwendungsfalldiagramm, verfeinert. Hierbei muss die durch das Personalisierungsframework zur Verfügung gestellte Navigations- und Verwaltungsfunktionalität (siehe Abschnitt 11.3.2) Berücksichtigung finden.

Das Ergebnis ist ein verfeinertes Anwendungsfalldiagramm (wie in Abschnitt 12.1 beschrieben), welches die Features der Applikation sowie deren Unterteilung in externe Dienste und Utility-Dienste (vgl. Kapitel 9) modelliert. Der Anwendungsexperte hat also schließlich die logisch in sich geschlossene Teilfunktionalitäten der Applikation, die Dienste, festgelegt und dabei für jeden Dienst aus der Anforderungsdefinition entschieden, welcher der folgenden drei Kategorien er zuzuordnen ist:

1. Das Personalisierungsframework stellt den Dienst bereits als Komponente zur Verfügung, und er kann ohne jegliche Änderung wiederverwendet werden.

   In diesem Fall ist also keine Entwicklungsarbeit für diesen Dienst zu leisten, sondern er muss lediglich vom Personalisierungsexperten im Rahmen der Domänenmodellierung für die Applikation, d.h. bei der Konfiguration des Personalisierungsframeworks, berücksichtigt werden.

2. Der Dienst steht als Komponente über das Personalisierungsframework zur Verfügung, es sind aber noch diverse Anpassungen erforderlich sind.

   In diesem Fall ist demnach noch in einigen Bereichen Entwicklungsabeit zu leisten, wie in Abb. 13.1 im hellgrau hinterlegten Bereich dargestellt. Dieser Bereich enthält dieselben Teilabläufe, wie der in Abb. 13.1 dunkelgrau hinterlegte Bereich, mit dem Unterschied, dass nicht zwingend alle Aktivitäten ausgeführt werden müssen, sondern nur die für die geforderten Änderungen bzw. Anpassungen notwendigen Aktivitäten.

3. Bei dem Dienst handelt es sich um einen applikations-spezifischen Dienst, der von Grund auf realisiert werden muss.

   Das hierfür notwendige Vorgehen ist in Abb. 13.1 im dunkelgrau hinterlegten Bereich dargestellt.

Beispiel 13.1:

Der Dienst *addRole*, welcher innerhalb der Bibliothek PwisUser definiert ist, bietet sich für die Wiederverwendung im TEMPLUS Projekt an. Er erlaubt es, Benutzer einer Benutzergruppe zuzuordnen. Bei Wiederverwendung dieses Dienstes ist eine Anpassung des entsprechenden Kontrollflusses bzgl. der möglichen Erfassung zusätzlicher, applikations-spezifischer (Rollen-)Daten notwendig.

Im Anwendungsbereich des Lehremanagements an einer Universität – umgesetzt im Rahmen des TEMPLUS Projektes – gibt es unterschiedliche Rollen, die keine zusätzliche Haltung von Rollendaten erfordern (z.B. Dozent, Tutor). Aber es gibt auch die Rolle Student, bei der für einen Benutzer, der diese Rolle innehat, zusätzliche Daten in Form von Matrikelnummer, Studiengang, Semester, etc. gespeichert werden müssen.

Will man nun den allgemeinen Dienst *addRole*, der keine Behandlung applikations-spezifischer Rollendaten beinhaltet, für eine Applikation wie TEMPLUS wiederverwenden, muss das Erfassen der Rollendaten für Benutzer der Rolle Student in diesen Dienst als Alternative im Kontrollfluss aufgenommen werden, damit die Applikation korrekt arbeiten kann.

⌟

Eine ausführliche Dokumentation für die identifizierten Dienste ist hierbei unbedingt notwendig, da andere Rollen diese Information als Ausgangspunkt für ihre Arbeitaufgaben benötigen – wie aus Abb. 13.1 ersichtlich ist. Der Personalisierungsexperte (siehe Abschnitt 13.1.3), der OO-Spezialist (siehe Abschnitt 13.1.4) und der GUI-Designer (siehe Abschnitt 13.1.7) benötigen diese Dokumentation für die Erfüllung ihrer Aufgaben.

Die identifizierten Dienste werden bzgl. ihrer Eigenschaften in die dem Projekt zugrunde liegende Taxonomie eingeordnet. Mögliche Eigenschaften für die Klassifizierung eines Dienstes sind beispielsweise:

- die Zugangsbeschränkung, d.h. ob es sich um einen öffentlichen Dienst, einen Login- bzw. Logout-Dienst oder einen privaten Dienst handelt
- das Eingabe-/Ausgabe-Verhalten, d.h. welche Daten als verfügbar vorausgesetzt werden und welche Daten der Dienst nach erfolgreicher Ausführung bereitstellt
- auf welchen Typen von Geschäftsobjekten der Dienst arbeitet

Einen vollständigen Überblick hierüber sowie über die Einsatzmöglichkeiten einer derartigen Klassifizierung der Dienste einer Applikation mittels einer Taxonomie vermittelt Abschnitt 10.2.

Jeder einzelne Dienst der Applikation und wird nun vom Anwendungsexperten von Grund auf als *SLG* modelliert (siehe Abb. 13.1, dunkelgrauer Bereich) oder durch Ausführen einzelner Aktivitäten entsprechend angepasst (siehe Abb. 13.1, hellgrauer Bereich).

Auf diese Weise wird die Dienstlogik festgelegt, d.h. die Abläufe, welche zur Durchführung der entsprechenden Dienstleistung notwendig sind. Im Einzelnen werden hier folgende Aktivitäten ausgeführt:

- Im ersten Schritt wird das entwickelte Zustandübergangsdiagramm für die GUI als initialer *SLG* importiert – wie in [11] beschrieben. Dieses Diagramm wird durch den System-Ingenieur in Absprache mit dem Kunden fertiggestellt, nachdem ein entsprechender GUI-Prototyp durch den GUI-Designer erstellt und präsentiert worden ist.
- Anschließend wird der *SLG* anhand der vorliegenden Beschreibung des zugehörigen Dienstes innerhalb des *ABC-SD* vervollständigt.
- Hierbei zusätzlich benötigte Komponenten werden durch den Komponenten-Integrator (siehe Abschnitt 13.1.6) umgesetzt. An ihn muss der Anwendungs-

experte die Schnittstellenbeschreibung sowie das intendierte Verhalten der benötigten Komponente weitergeben.

- Die Validierung eines Dienstes beinhaltet die Überprüfung lokaler sowie globaler Eigenschaften dieses Dienstes während der Modellierung, wie bereits in Abschnitt 5.2 kurz dargestellt, und ist der Schlüssel für die Entwicklung zuverlässiger, web-basierter Applikationen.
  Eine detaillierte Beschreibung und Klassifizierung der zu überprüfenden Eigenschaften sowie eine genaue Erklärung der Abläufe der im Rahmen dieses Entwicklungsprozess $MyProcess_{ABC\text{-}SD}$ zum Einsatz kommenden Validierungsmethoden befindet sich in Teil V.

- Bei der Modellierung der Koordinationsschicht des Dienstes verwendete Konfigurationsdaten werden dann in der zur Applikation gehörigen Konfigurationsdatei (`web.xml`), in der bereits der Personalisierungsexperte sowie der DB-Experte (siehe Abschnitt 13.1.5) Informationen eingetragen haben, ergänzt – wie in Kapitel 7 beschrieben.

- Wie in Abschnitt 5.3 beschrieben kann die Generierung der Koordinationsschicht für einen Dienst erfolgen, sobald die Validierung erfolgreich abgeschlossen ist.

- Im nächsten Schritt wird die Funktionalität des Dienstes getestet ([63, 45, 92, 41]).

- Die Information über die im *SLG* festgelegten Schnittstellen zur Präsentationsschicht der Applikation, d.h. konkret über den Typ und die Benennung der innerhalb der Präsentationsschicht verfügbaren Datensätze, muss an den GUI-Designer übermittelt werden.

- Der Integrationstest für einen Dienst komplettiert die bereits zuvor durchgeführten Testaktivitäten. U.a. liegt hierbei der Schwerpunkt bei dem funktionalen Testen auf GUI Ebene sowie den nicht-funktionalen Anforderungen an die Applikation (z.B. Performanz).

Sind die *SLG*s für die einzelnen Dienste fertiggestellt, folgt die Validierung von weiteren Eigenschaften. Diese beziehen sich auf die korrekte Hintereinanderausführung verschiedener Dienste. Eine detaillierte Beschreibung und Klassifizierung dieser Eigenschaften sowie eine genaue Erklärung der Abläufe befindet sich in Teil V.

Nach der Durchführung des Integrationstests für die gesamte Applikation, bei der jeder Dienst innerhalb des Frameworks und bzgl. seiner Wechselwirkung, Zusammenarbeit und Interaktion mit den anderen Diensten getestet wird, erfolgt dann das Bereitstellen der Applikation.

13.1.3 Personalisierungsexperte

Der Personalisierungsexperte erhält zunächst vom Anwendungsexperten (siehe Abschnitt 13.1.2) die Dokumentation der Dienste, welche innerhalb der Applikation realisiert, adaptiert bzw. wiederverwendet werden sollen.

Mithilfe der web-basierten Applikation, welche die Administrationsfunktionalität für das Personalisierungsframework (siehe Abschnitt 11.3.1) zur Verfügung stellt, nimmt der Personalisierungsexperte dann die Domänenmodellierung (siehe Abschnitt 12.4) für die Applikation vor.

Auf diese Weise wird u.a. festgelegt,

- welche Dienste die neue Applikation zur Verfügung stellt und
- wie diese Dienste angesprochen werden können.

Diese Informationen können im Anschluss zur Erstellung der initialen Konfigurationsdatei der Applikation verwendet werden, deren Aufbau in Abschnitt 7 erklärt wird und welche vom DB-Experten (siehe Abschnitt 13.1.5) und schließlich auch vom Anwendungsexperten in Abhängigkeit von den entwickelten *SLG*s für die einzelnen Dienste noch ergänzt werden muss.

13.1.4 OO-Spezialist

Die Geschäftsobjekte sowie ihre Verantwortlichkeiten und ihre Zusammenarbeit werden vom System-Ingenieur (siehe Abschnitt 13.1.1) auf konzeptioneller Ebene beschrieben. Dies bildet die Grundlage für die Aufgaben, welche der OO-Spezialist zu erfüllen hat.

Wie im Prozessmodell $PM_{ABC\text{-}SD}$ festgelegt, ist es auch hier Aufgabe des OO-Spezialisten anhand dieser Information, die Klassen für die Geschäftsobjekte des Anwendungsbereichs zu entwerfen, zu implementieren und zu testen.

Die übrigen Aufgaben des OO-Spezialisten ergeben sich durch den Einsatz des im Rahmen des *ABC-SD* zur Verfügung stehenden Persistenzframeworks[5] ([48]).

Dieser regelt die Speicherung persistenter Geschäftsobjekte in der Persistenzschicht der Applikation. Der Zugriff auf diese Geschäftsobjekte bzw. deren Modifikation erfolgt dabei mithilfe sogenannter DB-Administratoren. Ein solcher DB-Administrator

[5]Hierbei handelt es sich um Aufgaben, die sich aus den projekt-spezifischen Rahmenbedingungen herleiten. Sonst leistet an dieser Stelle innerhalb eines Entwicklungsprozesses oft ein O/R-Mapping Tool wertvolle Dienste.

verwaltet stets Geschäftobjekte eines Typs und stellt Funktionalitäten für Datenbankanfragen und -modifikationen bereit, die sich auf den zugehörigen Typ von Geschäftsobjekten beziehen. In den meisten Fällen erfüllt ein bereits realisierter Standard-Administrator diesen Zweck, der eine Menge häufig benötigter Funktionalitäten für die DB-Anfrage bzw. -Modifikation bereitstellt. Wo diese nicht genügen, können eigene DB-Administratoren realisiert werden.

Gerade bei komplexen Applikationen, die große Datenmengen verwalten, spielt die Art und Weise, wie auf diese Daten in der Persistenzschicht zugegriffen wird, eine entscheidende Rolle für die Antwortzeiten der Applikation bei der Bearbeitung einer Benutzeranfrage. Aufgabe des OO-Spezialisten ist es nun, für die persistenten Geschäftsobjekte die zugehörigen DB-Administratoren zu entwerfen, diese in Zusammenarbeit mit dem DB-Experten (siehe Abschnitt 13.1.5) zu implementieren und sie schließlich zu testen.

Der DB-Experte, der Komponenten-Integrator (siehe Abschnitt 13.1.6) und der GUI-Designer benötigen für die Erfüllung ihrer Aufgaben die Schnittstellenbeschreibung der Geschäftsobjekte sowie der zugehörigen DB-Administratoren vom OO-Spezialisten.

13.1.5 DB-Experte

Vom OO-Spezialisten (siehe Abschnitt 13.1.4) erhält der DB-Experte die Schnittstellenbeschreibung der Geschäftsobjekte sowie der zugehörigen DB-Administratoren. Auf Basis dieser entwirft und dokumentiert er die Struktur der Persistenzschicht der Applikation, d.h. der zugehörigen Datenbanktabellen, in Form einer XML-Datei.

Dokumentiert werden hierbei z.B. die Namen der verschiedenen Datenbanktabellen sowie für jede einzelne von ihnen folgende relevanten Eigenschaften:

- Name der Tabelle
- Klassenname des in der Tabelle zu speichernden Typs von Geschäftsobjekten
- Klassenname des DB-Administrators, welcher die in der Tabelle zu speichernden Geschäftsobjekte verwaltet
- Informationen über Beziehungen zu in anderen Datenbanktabellen gespeicherten Geschäftsobjekten
- Name, Typ sowie besondere Eigenschaften von Spalten

Die zugrunde liegende DTD ist in Anhang B enthalten.

Aus dieser XML-Datei zur Dokumentation der Datenbankstruktur können sowohl die entsprechenden SQL-Skripte, die zum Aufsetzen der Datenbank für die Applikation benötigt werden, als auch eine HTML-Version der Dokumentation generiert werden, die als Kommunikationsmedium im Kreis der am Entwicklungsprozess beteiligten Personen einsetzbar ist.

Mithilfe der automatisch generierten SQL-Skripte setzt der DB-Experte die Datenbank inklusive aller Tabellen für die Applikation auf. Die Namen der Datenbanktabellen trägt er in die Konfigurationsdatei der Applikation ein, welche vom Personalisierungsexperten (siehe Abschnitt 13.1.3) erstellt worden ist, und reicht diese an den Anwendungsexperten (siehe Abschnitt 13.1.2) weiter, welcher die Datei komplettiert. Aufgrund der automatischen Generierung der SQL-Skripte aus der XML Dokumentation für die Datenbankstruktur ist für das Aufsetzen der Datenbank nicht mehr zwingend ein Experte nötig, und die Fehleranfälligkeit ist geringer als bei einer manuellen Erstellung bzw. Anpassung der Datenbanktabellen.

Darüber hinaus unterstützt der DB-Experte den OO-Spezialisten bei der Implementierung der DB-Administratoren für die persistenten Geschäftsobjekte. Durch die Optimierung der SQL-Anfragen, welche in den Java-Code der innerhalb der DB-Administratoren realisierten Funktionalitäten eingebettet sind, werden die Antwortzeiten der Applikation bei der Bearbeitung von Benutzeranfragen minimiert und somit die Qualität der Applikation erhöht.

13.1.6 Komponenten-Integrator

Der Anwendungsexperte (siehe Abschnitt 13.1.2) fordert vom Komponenten-Integrator die Realisierung neuer, applikations-spezifischer Komponenten, die für das Erstellen der *SLG*s notwendig sind. Als Ausgangsbasis hierfür stellt der Anwendungsexperte dem Komponenten-Integrator eine genaue Beschreibung der Schnittstellen sowie des intendierten Verhaltens der benötigten Komponenten zur Verfügung.

Aufgabe des Komponenten-Integrators ist es nun, abzuwägen und dann zu entscheiden, ob eine Realisierung der benötigten Komponenten als *SIB* oder als Makro unter Verwendung bereits bestehender Komponenten eine adäquate Lösung darstellt.

Soll eine Komponente als *SIB* realisiert werden, erhält der Komponenten-Integrator vom OO-Spezialisten (siehe Abschnitt 13.1.4) die Schnittstellenbeschreibung der Geschäftsobjekte sowie der zugehörigen DB-Administratoren, auf denen die Komponente arbeitet, und kann bei Bedarf eine OO-Erweiterung anfordern.

Die Realisierung einer Komponente als Makro bietet einige Vorteile:

- Ein Makro wird als Graph unter Verwendung bestehender, getesteter, zuverlässig arbeitender Komponenten modelliert.

- Die Ablauflogik eines Makros wird durch die Struktur des zugehörigen Graphen umgesetzt und kann daher einfach und ohne Programmierkenntnisse geändert werden.
- Bei der Realisierung einer Komponente als Makro muss kein zusätzlicher Code geschrieben, getestet und gewartet werden.

Der Komponenten-Integrator muss also je nachdem,

- wie komplex eine Komponente ist,
- wie oft sie eingesetzt wird und
- wie groß die *SLG*s der Applikation werden dürfen,

entscheiden, auf welche Art und Weise er die benötigten Komponenten realisiert.

Die entwickelten Komponenten werden – wie zuvor die Dienste – anhand ihrer Eigenschaften in die dem Projekt zugrunde liegende Taxonomie eingeordnet. Mögliche Eigenschaften für die Klassifizierung von *SIB*s und Makros sind beispielsweise:

- die Art der ausgeführten Aktionen, d.h. ob die Komponente Benutzerinteraktion erlaubt oder lediglich systeminterne Aktionen durchgeführt werden,
- das Eingabe-/Ausgabe-Verhalten, d.h. welche Daten die Komponente in den vorhandenen Datenkontexten (siehe Kapitel 7) als verfügbar voraussetzt und welche Daten durch die Komponente in diesen Datenkontexten bereitgestellt werden, und
- auf welchen Typen von Geschäftsobjekten die Komponente arbeitet.

Einen vollständigen Überblick hierüber sowie über die Einsatzmöglichkeiten einer derartigen Klassifizierung von *SIB*s und Makros einer Applikation mittels einer Taxonomie vermittelt Abschnitt 10.1.

13.1.7 GUI-Designer

Das Zustandsübergangsdiagramm für die GUI Modellierung wird vom System-Ingenieur (siehe Abschnitt 13.1.1) mit dem Kunden abgestimmt und stellt die Ausgangsbasis für die Aufgaben des GUI-Designers dar. Der GUI-Designer erhält außerdem vom Anwendungsexperten (siehe Abschnitt 13.1.2) die Dokumentation der Dienste, welche innerhalb der Applikation realisiert werden sollen.

Zunächst erstellt er ausgehend vom Zustandsübergangsdiagramm für die GUI einen statischen Prototyp, welcher dazu dient, in Absprache und Diskussion mit dem Kunden herauszufinden, ob die GUI dessen Vorstellungen entspricht.

Das Zustandübergangsdiagramm legt die Interaktionspunkte einer web-basierten Applikation fest. Diese können dabei verschiedene Aufgaben erfüllen, z.B.:

- Startpunkt eines Dienstes
- Anzeigen von Daten
- Download von Daten

Beim Anzeigen bzw. Download von Daten kann eine web-basierte Applikation außerdem unterschiedliche Dateiformate unterstützen:

- Der Regelfall ist sicherlich eine HTML-Seite, die von der Applikation mit Daten gefüllt und anschließend dem Benutzer angezeigt wird.
- Daten können aber auch als normaler Text (Dateiformat ASCII-Text, Dateien mit Endung `.txt`) im Browser angezeigt werden.
- Darüber hinaus können Daten ohne Anzeigen direkt abgespeichert werden, z.B. in den Formaten CSV[6], XML[7], PDF[8] oder ASCII-Text (Endung `.txt`)

Der GUI-Designer muss demnach für jeden Interaktionspunkt, an dem Daten angezeigt oder abgespeichert werden sollen, einen entsprechenden Dateirahmen im geforderten Format für die zugehörige Datei realisieren, welcher dann zur Laufzeit von der Applikation mit den jeweiligen Daten gefüllt wird.

Vor der eigentlichen Implementierung dieser Dateirahmen (=Templates), muss der GUI-Designer das Zustandsübergangsdiagramm analysieren und häufig verwendete GUI-Elemente innerhalb der Präsentationsschicht der Applikation (wie z.B. eine Liste von Benutzern) identifizieren, die dann als wiederverwendbare, konfigurierbare Bausteine realisiert werden und bei der Erstellung der konkreten Templates zum Einsatz kommen. Auf diese Weise erhält man ein einheitliches Layout für alle Dienste der Applikation.

Die entwickelten Template-Bausteine werden – wie zuvor die Komponenten – anhand ihrer Eigenschaften in die dem Projekt zugrunde liegende Taxonomie eingeordnet. Mögliche Eigenschaften für die Klassifizierung von Template-Bausteinen sind

[6] Character Separated Values. Dateien mit Endung `.csv`
[7] Extensible Markup Language. Dateien mit Endung `.xml`
[8] Portable Document Format. Dateien mit Endung `.pdf`

beispielsweise das zugehörige Dateiformat sowie die Information, auf welche Typen von Geschäftsobjekten Bezug genommen wird. Diese Einordnung dokumentiert die Menge der vorhandenen Template-Bausteine, unterstützt Wiederverwendung und erleichert so die Arbeit des GUI-Designers.

Vom Anwendungsexperten wird dann die Information über die im *SLG* festgelegten Schnittstellen zur Präsentationsschicht der Applikation übermittelt, d.h. konkret über den Typ und die Benennung der durch die Applikation in den Templates verfügbar gemachten Datensätzen. Mit diesem Wissen kann der GUI-Designer nun die GUI integrieren und so die Templates fertigstellen. Daten werden dabei über eine entsprechende Skripting-Sprache in das Template eingebunden.

Die Templates werden schließlich vom Anwendungsexperten im Rahmen des Integrationstests auf mögliche Fehler überprüft.

13.2 Abhängigkeiten

Durch die in Abschnitt 13.1 dargestellte Arbeitsteiligkeit innerhalb des Entwicklungsprozesses $MyProcess_{ABC\text{-}SD}$ sind Schnittstellen zwischen den verschiedenen Rollen und deren Arbeitsbereichen vorgegeben. Demnach hängen die Entwicklungsergebnisse voneinander ab bzw. bauen aufeinander auf. Die Konsistenz dieser muss sichergestellt werden, damit die arbeitsteilig entwickelten Bereiche einer Applikation zusammen passen und somit diese Applikation schließlich zuverlässig funktionieren kann.

Abb. 13.2 präsentiert eine Übersicht und Klassifizierung der bestehenden Abhängigkeiten. Dabei wird jeweils die Art der Abhängigkeit in Bezug gesetzt zu den betroffenen Rollen.

Die folgenden Abschnitte beschreiben die bestehenden Schnittstellen und Abhängigkeiten detaillierter.

13.2.1 Applikation

Einen Überblick über die Gesamtfunktionalität der web-basierten Applikation haben die am Entwicklungsprozess beteiligten Rollen System-Ingenieur, Anwendungsexperte und Personalisierungsexperte. Dabei ist allerdings die Granularität bei der Betrachtung unterschiedlich. Abb. 13.3 veranschaulicht die Zuständigkeitsbereiche der verschiedenen Rollen.

Der System-Ingenieur erstellt in Zusammenarbeit mit dem Kunden eine Geschäftsprozessdefinition, welche die Gesamtfunktionalität der zu entwickelnden Applikation ab-

		System-Ingenieur	Anwendungsexperte	Personalisierungsexperte	DB-Experte	OO-Spezialist	Komponenten-Integrator	GUI-Designer
(1)	*Applikation / Gesamtfunktionalität*	×	×					
(2)	*Applikation / Konfiguration*		×	×	×			
(3)	*Applikation / Intitialisierung*		×		×	×		
(4)	*Geschäftsobjekte / Typ*		×			×		
(5)	*Geschäftsobjekte / Methoden*		×			×	×	×
(6)	*Geschäftsobjekte / Persistenz*				×	×		
(7)	*Geschäftsobjekte / Assoziationen*				×	×		
(8)	*Geschäftsobjekte / Administratoren*				×	×		
(9)	*Präsentation / Benutzerinteraktion*		×					×
(10)	*Präsentation / Dateien*		×					×
(11)	*Präsentation / Daten*		×					×
(12)	*Präsentation / Kontrollfluss*		×					×
(13)	*Komponenten / Administratoren*		×			×	×	
(14)	*Komponenten / Spezifikation*		×				×	

Abbildung 13.2: Abhängigkeiten im Entwicklungsprozess $MyProcess_{ABC\text{-}SD}$

bildet, sowie ein initiales Anwendungsfalldiagramm. Auf Basis der Geschäftsprozesse verfeinert der Anwendungsexperte das Anwendungsfalldiagramm unter Berücksichtigung des Personalisierungsframeworks. Dies umfasst die Identifikation der Features bzw. der in diesen enthaltenen Diensten, eine logische Gruppierung dieser sowie die Zuordnung der Features zu den im Rahmen der Applikation benötigten Rollen. Der Personalisierungsexperte erstellt dann auf Grundlage dieser Informationen die Konfigurationsdatei der Applikation und initialisiert das Personalisierungsframework, während der Anwendungsexperte für die eigentliche Realisierung der Dienste verantwortlich ist.

Aus diesen Zusammenhängen ergeben sich nun einige Abhängigkeiten, die in den folgenden Abschnitten näher erläutert werden.

(1) Gesamtfunktionalität

Die Geschäftsprozesse einer web-basierten Applikation werden durch den System-Ingenieur definiert. Auf Basis dieser Definition der einzelnen Geschäftprozesse nimmt

$$\text{Gesamtfunktionalität} \supseteq \text{Geschäftsprozess} \supseteq \text{Feature} \supseteq \text{Dienst}$$

System-Ingenieur

Anwendungsexperte

Personalisierungsexperte

Abbildung 13.3: Zuständigkeitsbereiche der verschiedenen Rollen

dann der Anwendungsexperte eine Aufteilung der Gesamtfunktionalität der Applikation in Features und schließlich in Dienste vor. Die Menge der auf diese Art und Weise identifizierten Dienste muss dabei immer – d.h. sowohl beim initialen Durchlauf des Prozessmodells als auch bei einer inkrementellen Erweiterung der Applikation – konsistent mit den vorgegebenen Geschäftsprozessen sein, damit das fertige Produkt schließlich den Anforderungen des Kunden entspricht.

(2) Konfiguration

Die Konfiguration einer mit *ABC-SD* erstellten personalisierten, web-basierten Applikation erfolgt – wie bereits in Kapitel 7 beschrieben – mithilfe der Konfigurationsdatei `web.xml`, die festlegt, welche Dienste zur Applikation gehören, auf dem Server zur Verfügung stehen und ausgeführt werden dürfen. In dieser Datei werden daher alle Dienste der Applikation definiert, aus denen sich die Features zusammensetzen können.

Am Entwicklungsprozess beteiligte Personen der Rollen Personalisierungsexperte und Anwendungsexperte sind an der Erstellung und Modifikation dieser Konfigurationsdatei beteiligt. Darüber hinaus ergänzt der DB-Experte hier die Namen der zu den Geschäftsobjekten der Applikation gehörigen Datenbanktabellen.

Der Personalisierungsexperte konfiguriert außerdem über eine web-basierte Applikation, welche Dienste diese zur Verfügung stellen soll. Dabei werden z.B. der Name des Dienstes sowie dessen URL festgelegt und implizit die zugehörigen Einträge in der Datenbank angelegt bzw. modifiziert.

Dabei bestehen folgende Abhängigkeiten:

- Die URL des zu einem Feature gehörigen Dienstes muss in der `web.xml` sowie während der Konfiguration durch den Personalisierungsexperten eingetragen und als Name des entsprechenden *SLG*s gesetzt werden.
- Die durch den Eintrag in die Datenbank automatisch festgelegte eindeutige

Identifizierung muss zur Realisierung der Zugriffberechtigungsprüfung innerhalb des *SLG*s verwendet werden.

(3) Initialisierung

Die Geschäftsobjekte sowie die zugehörigen Datenbankadministratoren werden durch Entwickler der Rollen OO-Spezialist und DB-Experte realisiert. Der DB-Experte setzt darüber hinaus die Datenbank für die Applikation auf, legt also u.a. die Namen der zu den Geschäftsobjekten gehörigen Tabellen fest.

Diese Informationen müssen dem Anwendungsexperten bei der Erstellung der Koordinationsschicht zur Verfügung stehen, da hier auch der Zugang zur Datenbank und deren Tabellen initialisiert wird. Im Einzelnen sind dort die Datenbankzugangsparameter, Parameter zu Geschäftsobjekten (z.B. Klassenpfad) sowie der zugehörigen Datenbank-Administratoren (Klassenpfad, Tabellenname, Keyattribut für die zugehörige Tabelle etc.) einzutragen.

13.2.2 Geschäftsobjekte

Die Geschäftsobjekte und deren Beziehungen zueinander beschreiben den Anwendungsbereich einer Applikation.

(4) Typ

Die verschiedenen Typen von Geschäftsobjekten werden vom OO-Spezialisten aus der konzeptuellen Beschreibung, die er vom System-Ingenieur erhält, extrahiert bzw. identifiziert und anhand der Featurebeschreibung, die er vom Anwendungsexperten erhält, entsprechend verfeinert. Sie bilden die Basis für die gesamte Entwicklungsarbeit, da sie den Anwendungsbereich der Applikation modellieren.

(5) Methoden

Die Geschäftsobjekte stellen Methoden zur Verfügung, deren Signatur der OO-Spezialist während der Implementierung festlegt. Diese Schnittstelleninformation muss folgenden anderen Entwicklern bekannt sein, die mit den Geschäftobjekte im Rahmen der Erfüllung ihrer Entwicklungsaufgaben umgehen:

- Der Komponenten-Integrator benutzt die Geschäftsobjekte bei der Implementierung neuer *SIB*s und muss daher die bereitgestellten Methoden und deren Signatur kennen.

- Bei der Erstellung der Templates für die Präsentationsschicht wird auf Daten, d.h. Geschäftsobjekte des Anwendungsbereichs zugegriffen. Sowohl der GUI-Designer als auch der Anwendungsexperte, der die Daten innerhalb der Präsentationsschicht verfügbar macht, müssen die Methoden-Signaturen kennen, um die korrekte und vollständige Bereitstellung benötigter Daten gewährleisten zu können.

(6) Persistenz

Ein persistentes Geschäftsobjekt wird innerhalb der Persistenzschicht der Applikation gespeichert, hat also eine zugehörige DB-Tabelle. Die zu speichernden Attribute eines Geschäftsobjekts werden dabei vom OO-Spezialisten während der Implementierung festgelegt, während der DB-Experte die zugehörige Datenbanktabelle bereitstellt.

Wichtig beim Einsatz des Persistenzframeworks im Rahmen des *ABC-SD* ist hierbei, dass es für jedes zu speichernde Attribut eines Geschäftsobjekts eine Spalte gleichen Namens in der zugehörigen Datenbanktabelle geben muss und umgekehrt. Außerdem muss für jedes zu speichernde Attribut der in der Implementierung der Klasse verwendete Typ dem in der Datenbank definiertem Typ der zugehörigen Spalte entsprechen und umgekehrt.

(7) Assoziationen

Die Assoziationsstruktur im Klassendiagramm der Applikation ermöglicht das Navigieren zwischen den zugehörigen Geschäftsobjekten. Bei persistenten Geschäftsobjekten spiegelt sich diese Verlinkung auf Datenbankebene wieder, wo dadurch Verbundanfragen ermöglicht werden.

Beispiel 13.2:

Betrachten wir die persistenten Geschäftsobjekte 'Lehrveranstaltung' (Klasse `Course`) und 'Gruppe einer Lehrveranstaltung' (Klasse `Group`) aus dem TEMPLUS Projekt: Geschäftsobjekte vom Typ `Group` kennen die ID ihrer zugehörigen Lehrveranstaltung (Methode `getCourseKey()`) und bieten die Methode `getCourse()` an, um zu dieser zu navigieren. Auf Datenbankebene bedeutet dies, dass in der Tabelle `groups4course` für Objekte vom Typ `Group` der Fremdschlüssel `courseKey`, d.h. die eindeutige ID der Tabelle `course`, gehalten wird, der das zugehörige Objekt vom Typ `Course` eindeutig identifiziert. Auf diese Weise werden Verbundanfragen möglich, z.B. die Ermittlung aller zu einer Lehrveranstaltung gehörigen Gruppen. ⌟

DB-Experte und OO-Spezialist müssen demnach Informationen über die Menge solcher Assoziationsstrukturen austauschen, da diese eine wichtige Rolle für die Effizienz einer Applikation spielen können.

(8) Administratoren

Persistente Geschäftsobjekte werden mithilfe von Datenbankadministratoren verwaltet, welche die Kommunikation und den Datenaustausch zwischen der Koordinations- und Persistenzschicht einer Applikation garantieren. Dies ist durch den Einsatz des Persistenzframeworks im *ABC-SD* bedingt.

Dabei gibt es für jeden Typ von Geschäftsobjekten ein Paar von Datenbankadministratoren:

- Der *Smart-Administrator* wird durch den OO-Spezialisten implementiert, steht innerhalb der Koordinationsschicht einer Applikation zur Verfügung und bietet komfortable Methoden für die Verwaltung der Geschäftsobjekte.
- Der *JDBC-Administrator* wird durch den DB-Spezialisten implementiert, kann nur indirekt über den Smart-Administrator angesprochen werden und enthält die Implementierung applikations-spezifischer Datenbankoperationen unter Verwendung von SQL.

OO-Spezialist und DB-Experte müssen demnach Informationen zu den Schnittstellen zwischen Smart-Administratoren und JDBC-Admininistratoren austauschen, damit die Applikation zuverlässig und effizient arbeiten kann.

13.2.3 Präsentation

In diesem Abschnitt werden die Abhängigkeiten erläutert, die bei einer web-basierten Applikation zwischen der durch den Anwendungsexperten entwickelten Koordinationsschicht und der durch den GUI-Designer realisierten Präsentationsschicht bestehen.

(9) Benutzerinteraktion

Durch das vom System-Ingenieur erstellte Zustandübergangsdiagramm für die GUI wird eine bestimmte Abfolge von Interaktionspunkten (i.d.R. handelt es sich hierbei um HTML-Seiten) für eine web-basierte Applikation vorgegeben. Diese Navigationsstruktur spiegelt sich in dem initialen *SLG* wieder, der durch Importieren dieses Diagramms entsteht und dann vom Anwendungsexperten verfeinert wird.

Da am Entwicklungsprozess beteiligte Personen dieser beiden Rollen nach dem Austausch dieser Informationen zunächst unabhängig voneinander weiterarbeiten und dabei auch Änderungen vornehmen können, muss später bei der Integration von GUI und Applikationslogik sichergestellt werden können, dass die Entwicklungsergebnisse zusammenpassen.

(10) Dateien

Den durch das Zustandsübergangsdiagramm für die GUI beschriebenen Punkte mit Benutzerinteraktion einer web-basierten Applikation kann innerhalb der Koordinationsschicht, d.h. über einen Parameter spezieller *SIB*s mit Benutzerinteraktion aus dem *SLG*, durch Angabe eines entsprechenden Dateinamens das zu verwendende Template zugeordnet werden. Eine eindeutige Zuordnung und Benennung ist dabei unbedingt erforderlich für die korrekte Funktionsweise und Zuverlässigkeit der Applikation.

(11) Daten

Benutzerinteraktion im Rahmen einer web-basierten Applikation beinhaltet auch den Datenaustausch zwischen ihrer Koordinations- und Präsentationsschicht.

Zum einen werden durch den Anwendungsexperten die zu präsentierenden Daten im Rahmen des *SLG*s bereitgestellt, so dass der GUI-Designer auf diese Daten innerhalb der Templates zugreifen kann.

Zum anderen werden durch zusätzliche Parameter im Rahmen einer Benutzeranfrage Daten an die Koordinationsschicht der Applikation übermittelt, auf die der Anwendungsexperte bei der Erstellung des *SLG* zugreifen kann.

Eine eindeutige Benennung der auszutauschenden Daten ist dabei unbedingt notwendig. Die Prüfung der Konsistenz verhindert, dass zur Laufzeit Probleme aufgrund fehlender Daten auftreten.

(12) Kontrollfluss

Neben den Interaktionspunkten legt das Zustandsübergangsdiagramm für die GUI auch die Navigationsstruktur, d.h. den Kontrollfluss, einer Applikation fest. Diese Information wird an zwei Stellen verwendet:

- Der GUI-Designer muss die Navigationsstruktur mithilfe von Links und Formularen auf Ebene der HTML-Templates realisieren.

- Der Anwendungsexperte muss im *SLG* durch Benennung der von Interaktionspunkten ausgehenden Kanten den Kontrollfluss abbilden.

Eine eindeutige, konsistente Benennung ist dabei notwendig, damit die Applikation korrekt funktionieren kann.

13.2.4 Komponenten

(13) Administratoren

Datenbankadministratoren stellen eine Menge von Methoden für die Verwaltung persistenter Geschäftsobjekte zur Verfügung. Diese Schnittstelleninformation muss sowohl dem Komponenten-Integrator für die Implementierung von *SIB*s, als auch dem Anwendungsexperten für die Realisierung der Applikationslogik bekannt sein.

(14) Spezifikation

Der Anwendungsexperte definiert die Schnittstellen sowie das intendierte Verhalten von neuen Komponenten, die er benötigt, um den *SLG* eines Features der Applikation zu erstellen und leitet diese Spezifikation an den Komponenten-Integrator weiter, der die neue Komponente nach eben diesen Anforderungen realisiert. Bevor schließlich die Applikation generiert werden kann, muss sichergestellt sein, dass die Spezifikation und die Implementierung der jeweiligen Komponente miteinander konsistent sind.

Teil V

Framework – Validierung

Kapitel 14

Validierung – Ein Überblick

Die Koordinationsschicht ist bei der Entwicklung personalisierter, web-basierter und als Dienstfamilie konzipierter Applikationen mit *ABC-SD* zentral, wie bereits aus der Beschreibung des Entwicklungsprozesses $MyProcess_{ABC\text{-}SD}$ (siehe Kapitel 13) hervorgeht.

In den Händen des Applikationsexperten laufen dabei alle Fäden zusammen, d.h. er zeichnet verantwortlich für das Zusammenfügen der Teilergebnisse, die im Rahmen der rollen-basierten, arbeitsteiligen Entwicklung der Applikation entstehen.

Eine wesentliche Aufgabe des Applikationsexperten ist die Validierung. Mithilfe dieser wird vor dem Bereitstellen die Zuverlässigkeit und Konsistenz der entwickelten Applikation überprüft.

Dem Applikationsexperten stehen im Rahmen der Validierung verschiedene Mechanismen und Methoden zur Verfügung, mit denen er einen Teil der durch die rollen-basierte, arbeitsteilige Entwicklung einer web-basierten Applikation bedingten Abhängigkeiten (siehe Abschnitt 13.2) geeignet prüfen kann. Auf diese Weise kann die Zuverlässigkeit der Workflows einer Applikation erhöht werden.

Wir unterscheiden zwei Arten der Validierung, die bereits in Abschnitt 5.2 beschrieben sind:

- *Local Check* (*LC*) dient zur Überprüfung lokaler Eigenschaften (siehe Kapitel 15).
- Modelchecking (*MC*) wird für die Überprüfung globaler Eigenschaften eingesetzt (siehe Kapitel 16).

Die folgende Tabelle gibt einen Überblick darüber, welche dieser Abhängigkeiten im Rahmen der Validierung sinnvoll behandelt und automatisch überprüft werden können. Die Nummerierung der Abhängigkeiten entspricht den Nummern, die in Abschnitt 13.2 bei der Beschreibung der Abhängigkeiten verwendet worden sind.

Darüber hinaus wird den Abhängigkeiten, die im Rahmen der in den folgenden Kapiteln beschriebenen Validierungsmöglichkeiten behandelt werden, in der Tabelle jeweils eine adäquate Validierungsmethode zugeordnet.

Abhängigkeit (siehe Abschnitt 13.2)		***Local Check***	**Modelchecking**
(1)	*Applikation / Gesamtfunktionalität*	–	–
(2)	*Applikation / Konfiguration*	× (Abschnitt 15.2.3)	–
(3)	*Applikation / Intitialisierung*	–	× (Anhang C.1.2)
(4)	*Geschäftsobjekte / Typ*	–	–
(5)	*Geschäftsobjekte / Methoden*	–	–
(6)	*Geschäftsobjekte / Persistenz*	–	–
(7)	*Geschäftsobjekte / Assoziationen*	–	–
(8)	*Geschäftsobjekte / Administratoren*	–	–
(9)	*Präsentation / Benutzerinteraktion*	–	–
(10)	*Präsentation / Dateien*	× (Abschnitt 15.2.4)	–
(11)	*Präsentation / Daten*	–	× (Abschnitt 16.4)
(12)	*Präsentation / Kontrollfluss*	× (Abschnitt 15.2.1)	× (Abschnitt 16.4) (Anhang C)
(13)	*Komponenten / Administratoren*	–	–
(14)	*Komponenten / Spezifikation*	× (Abschnitt 15.2.2)	–

Abbildung 14.1: Abhängigkeiten und Validierung

Kapitel 15

Überprüfung lokaler Eigenschaften

Lokale Eigenschaften werden im *ABC-SD* bei der Betrachtung einer einzelnen Komponente überprüft ohne das Umfeld zu beachten, in das die Komponente eingebettet ist. Entsprechend der drei verschiedenen Arten von Komponeten – *SIB*s, Makros und Dienste – werden hierbei folgende Szenarien unterschieden:

- Für *SIB*s, die im Rahmen von Makros und *SLG*s verwendet werden, können beispielsweise auf Basis der Eingabe- und Ausgabeparameter sowie der Branches verschiedene Eigenschaften geprüft werden, ohne andere Komponenten oder den Bezug des aktuellen *SIB*s zu diesen innerhalb des Makros oder *SLG*s zu berücksichtigen.

  Beispiel 15.1:

  Der *SIB* `ShowFile` aus der Bibliothek *EWIS* Base dient dem Anzeigen einer HTML-Seite und besitzt daher einen *SIB*-Parameter, welcher den Namen der zugehörigen HTML-Datei angibt. Eine überprüfbare lokale Eigenschaft für diesen *SIB*-Typ ist die Existenz der zugehörigen HTML-Datei. ⌟

- Makros können im Rahmen von anderen Makros bzw. *SLG*s verwendet werden. Bei einer Überprüfung lokaler Eigenschaften eines Makros, wird dieses als Blackbox betrachtet, d.h. der es definierende Graph wird nicht expandiert.

  Beispiel 15.2:

  Das Makro `CheckAuthorizationContent` ist in der Bibliothek PWISBase definiert und realisiert die Zugriffsberechtigungsprüfung. Hierfür gibt es einen Eingabeparameter `feature_id`, der gesetzt sein muss, damit das Makro korrekt funktionieren kann. Dies wird als lokale Eigenschaft geprüft. ⌟

- Dienste sind Teil eines Workflows einer Applikation. Im Rahmen eines Workflows können lokale Eigenschaften von Diensten geprüft werden. Dabei werden die Dienste – wie zuvor die applikations-spezifischen Makros – als Blackbox betrachtet, d.h. der zugehörige *SLG* wird nicht expandiert.

  Beispiel 15.3:

  Für einen Dienst kann die Existenz der zugehörigen Spezifikation, d.h. des zugehörigen *SLG*s, sowie der Implementierung des Dienstes, d.h. der generierten JAVA-Dateien, als lokale Eigenschaft geprüft werden. ⌟

In Abschnitt 15.1 werden die Möglichkeiten bei der Überprüfung lokaler Eigenschaften vor Einführung des Frameworks dargestellt.

Abschnitt 15.2 enthält einen Katalog lokaler Eigenschaften, welche alle durch die arbeitsteilige, rollen-basierte Entwicklung bedingten Abhängigkeiten abdecken, die lokal geprüft werden können.

15.1 Lokale Eigenschaften für *SIB*s

Lokale Eigenschaften für *SIB*s werden in der Interpreter-Sprache *HLL* als *Local Check Code* (*LCC*) spezifiziert [38, 37] und bei Betätigen des *Local Check* (*LC*) Buttons innerhalb des *SLG*- bzw. Makro-Fensters im *ABC-SD* überprüft.

Die Dateien, die den *Local Check Code* enthalten, müssen im `sib` Verzeichnis des *ABC*-Projektes platziert sein, welches den jeweiligen *SIB* über eine Bibliothek bereitstellt. Dabei gibt es für jeden *SIB* drei Arten von *LCC*, der bei der Überprüfung lokaler Eigenschaften berücksichtigt wird:

- **Globale Eigenschaft**:
  Die Datei `_global.lcc` enthält die Spezifikation des *LCC*, der für alle *SIB*s ausgeführt wird.

- **Eigenschaft für einen speziellen *SIB***:
  Für jeden *SIB*-Typ kann darüber hinaus individueller *Local Check Code* geschrieben werden. Dieser wird dann in einer Datei `<sib_id>.lcc` bereitgestellt, wobei `<sib_id>` den Typ des *SIB*s angibt. Dieser *Local Check Code* wird dann für jede Instanz dieses *SIB*-Typs ausgeführt.

- **Eigenschaften für alle *SIB*s einer *SIB*-Klasse**:
  Für jede *SIB*-Klasse kann eine Datei `<sib_class>.class.lcc` definiert wer-

den, die *Local Check Code* enthält, welcher für alle *SIB*s ausgeführt wird, die zu dieser *SIB*-Klasse gehören.

Für jeden *SIB* ist die *SIB*-Klasse frei wählbar, ja sogar änderbar. Diese Möglichkeit, *Local Check Code* zu schreiben, ist also sehr fehleranfällig und sollte daher nicht verwendet werden.

Beispiel 15.4:

Der folgende *Local Check Code* identifiziert isolierte *SIB*s, d.h. Knoten im Graphen, die weder eingehende noch ausgehende Kanten haben und somit vom Kontrollfluss nie sinnvoll erreicht werden können.

```
(* check for stray nodes which have nor ingoing edges
   neither outgoing edges *)
if ((PLGraph.indeg (SDLocalCheck.local_check_node) == 0) and
    (PLGraph.outdeg (SDLocalCheck.local_check_node) == 0)) then
   SDLocalCheck.localCheckError ("Stray node found.");
fi;
```

Hierbei handelt es sich um eine lokale Eigenschaft, die für alle *SIB*s überprüft werden muss. ⌟

15.2 Katalog Lokaler Eigenschaften

Wir unterscheiden drei Typen von lokalen Eigenschaften:

- Typ (1): ***global***, d.h. für alle Komponenten
- Typ (2): ***für eine konkrete Komponente***
- Typ (3): ***für eine Menge von Komponenten***

Die folgenden Abschnitte präsentieren einen Katalog lokaler Eigenschaften, die gemäß den vier Typen von Abhängigkeiten, welche das Prozessmodell impliziert und die mithilfe lokaler Eigenschaften überprüft werden können, gruppiert sind – wie bereits in Kapitel 14 dargestellt.

15.2.1 Kontrollfluss

Der Kontrollfluss eines Dienstes einer mit dem *ABC-SD* entwickelten web-basierten Applikation wird durch die Kanten in dem zu dem Dienst gehörigen *SLG* beschrieben (siehe Kapitel 5). Durch die Definition und Überprüfung lokaler Eigenschaften, welche sich auf den Kontrollfluss einer Applikation beziehen, können Fehler im Graphaufbau frühzeitig während der Entwicklung aufgedeckt und behoben werden.

Folgende lokalen Eigenschaften müssen in Bezug auf den im Rahmen einer HTML-Seite definierten Kontrollfluss einer mit dem *ABC-SD* entwickelten web-basierten Applikation gelten:

- ***Benutzerinteraktion*** im Rahmen einer web-basierten Applikation findet über HTML-Seiten statt. Dort kann ein Benutzer Links verfolgen bzw. Buttons anklicken, um mit der Applikation zu interagieren.

  Innerhalb des Kontrollflusses wird das Anzeigen einer HTML-Seite mithilfe eines `ShowFile` [1] *SIB*s realisiert.

  Es besteht also eine Abhängigkeit zwischen der Präsentationsschicht (konkret: einer HTML-Seite) der Applikation und ihrer Koordinationsschicht (konkret: dem `ShowFile` *SIB* im Graphen).

  Die lokale Eigenschaft $LcProp_1$, welche hier geprüft werden muss, ist vom Typ (2) und lautet:

  'Jede von einem `ShowFile` *SIB ausgehende Kante in einem SLG oder Makro entspricht immer einem Link bzw. Button in der zu diesem SIB gehörigen HTML-Seite und umgekehrt.'*

  Die Menge $LcNodes_1$ der Komponenten, für die diese Eigenschaft geprüft werden muss, kann innerhalb der Taxonomie folgendermaßen beschrieben werden:

  $$LcNodes_1 =_{df} SIB \cap \texttt{ShowFile}$$

  Die zugehörige lokale Eigenschaft $LcProp_1$ kann auf folgende Art und Weise formal definiert werden:

  $$LcProp_1 =_{df} \forall \ \text{node} \in LcNodes_1. \\ (\text{LcUtil.checkBranchLinkMappingOk(node)})$$

[1] Dieser *SIB* ist in der Bibliothek *EWIS* Base definiert.

- Bestimmte Komponenten kennzeichnen das **Ende des Kontrollflusses**. Beispielsweise können mithilfe eines `DownloadFile` [2] *SIB*s einem Benutzer andere Dokumentenformate als HTML zum Herunterladen angeboten werden, ein `ServiceCall` [2] *SIB* ermöglicht die Übergabe des Kontrollflusses an einen anderen Dienst. Somit endet in diesen Fällen der Kontrollfluss innerhalb des aktuellen *SLG*s.

  Da der Kontrollfluss an diesen ausgezeichneten Stellen endet, dürfen die zugehörigen Komponenten keine ausgehenden Kanten besitzen.

  Die lokale Eigenschaft $LcProp_2$, welche hier geprüft werden muss, ist vom Typ (3) und lautet:

  'Komponenten, an denen der Kontrollfluss endet, dürfen keine ausgehenden Kanten besitzen.

  Die Menge $LcNodes_2$ der Komponenten, für die diese Eigenschaft geprüft werden muss, kann innerhalb der Taxonomie folgendermaßen beschrieben werden:

  $$LcNodes_2 =_{df} \text{Component} \cap \text{End}$$

  Die zugehörige lokale Eigenschaft $LcProp_2$ kann auf folgende Art und Weise formal definiert werden:

  $$LcProp_2 =_{df} \forall \;\; \text{node} \in LcNodes_2.\ (\text{LcUtil.OutdegZero(node)})$$

- Ein ***unerreichbarer Teilbaum*** liegt vor, wenn für einen Knoten alle Branchnamen an die von ihm ausgehenden Kanten vergeben sind, aber dennoch eine unbeschriftete Kante existiert. Dann sind innerhalb des Teilbaums, der durch Verfolgen der unbeschrifteten Kante erreicht wird, alle Knoten, zu denen man ausschließlich über die unbeschriftete Kante gelangt, durch den Kontrollfluss nicht erreichbar.

  Die lokale Eigenschaft $LcProp_3$, welche hier geprüft wird, ist vom Typ (1) und lautet:

  'Ein Graph darf keine unerreichbaren Teilbäume enthalten.'

  Die Menge $LcNodes_3$ der Komponenten, für die diese Eigenschaft geprüft werden muss, kann innerhalb der Taxonomie folgendermaßen beschrieben werden:

  $$LcNodes_3 =_{df} \text{Component}$$

[2]Dieser *SIB* ist in der Bibliothek PwisBase definiert.

Die zugehörige lokale Eigenschaft $LcProp_3$ kann auf folgende Art und Weise formal definiert werden:

$$LcProp_3 =_{df} \forall \ \text{node} \in LcNodes_3. \ (\text{LcUtil.markDeadSibs(node)})$$

Wird ein derartiger Konfigurationsfehler bei der lokalen Überprüfung mit *ABC-SD* entdeckt, werden sowohl der Knoten, von dem die unbeschriftete Kante ausgeht, als auch alle Knoten, die in dem unerreichbaren Teilbaum liegen, blau markiert, um den entdeckten Fehler zu kennzeichnen.

- ***Unbeschriftete Kanten*** definieren Pfade, die vom Kontrollfluss nie erreicht werden können.

  Die lokale Eigenschaft $LcProp_4$, welche hier geprüft wird, ist vom Typ (1) und lautet:

  'Ein Graph darf keine unbeschrifteten Kanten enthalten.'

  Die Menge $LcNodes_4$ der Komponenten, für die diese Eigenschaft geprüft werden muss, kann innerhalb der Taxonomie folgendermaßen beschrieben werden:

  $$LcNodes_4 =_{df} \text{Component}$$

  Die zugehörige lokale Eigenschaft $LcProp_4$ kann auf folgende Art und Weise formal definiert werden:

  $$LcProp_4 =_{df} \forall \ \text{node} \in LcNodes_4. \ (\text{LcUtil.hasNoUnlabelledBranches(node)})$$

- Ein ***isolierter Knoten*** besitzt weder eingehende noch ausgehende Kanten und kann somit durch den Kontrollfluss nie sinnvoll erreicht werden.

  Die lokale Eigenschaft $LcProp_5$, welche hier geprüft wird, ist vom Typ (1) und lautet:

  'Ein Graph darf keine isolierten Knoten enthalten.'

  Die Menge $LcNodes_5$ der Komponenten, für die diese Eigenschaft geprüft werden muss, kann innerhalb der Taxonomie folgendermaßen beschrieben werden:

  $$LcNodes_5 =_{df} \text{Component}$$

  Die zugehörige lokale Eigenschaft $LcProp_5$ kann auf folgende Art und Weise formal definiert werden:

$$LcProp_5 =_{df} \forall \text{ node} \in LcNodes_5.\ (\text{LcUtil.NotIndegAndOutdegZero(node)})$$

- Bei der Modellierung der Koordinationsschicht einer web-basierten Applikation mit *ABC-SD* ist es möglich, einen ***unzusammenhängenden Graphen*** zu spezifizieren. Allerdings gelten einige Einschränkungen, damit die Applikation dennoch korrekt funktionieren kann.

  Lediglich *SIB*s vom Typ `Start` [3] oder `ServiceEntry` [3] können im Rahmen einer web-basierten Applikation vom Benutzer direkt über den Browser durch Eingabe einer URL angesprochen werden. Diese *SIB*s benötigen also keine eingehende Kante, um ausgeführt zu werden. Alle anderen *SIB*-Typen können vom Kontrollfluss nicht erreicht und ausgeführt werden, wenn sie keine eingehende Kante besitzen.

  Die lokale Eigenschaft $LcProp_6$, welche hier geprüft wird, ist vom Typ (3) und lautet:

  'Alle SIBs, die nicht vom Typ `Start` *oder* `ServiceEntry` *sind, müssen mindestens eine eingehende Kante besitzen.'*

  Die Menge $LcNodes_6$ der Komponenten, für die diese Eigenschaft geprüft werden muss, kann innerhalb der Taxonomie folgendermaßen beschrieben werden:

  $$LcNodes_6 =_{df} (\text{SIB} \cap \neg\ (\texttt{Start} \cup \texttt{ServiceEntry}))$$

  Die zugehörige lokale Eigenschaft $LcProp_6$ kann auf folgende Art und Weise formal definiert werden:

  $$LcProp_6 =_{df} \forall \text{ node} \in LcNodes_6.\ (\text{LcUtil.hasIndeg(node)})$$

- Jeder Dienst wird durch einen *SLG* realisiert. Damit dieser korrekt funktionieren kann, muss immer ein *SIB* vom Typ `Start` der ***Startknoten des zugehörigen SLGs*** sein.

  Die lokale Eigenschaft $LcProp_7$, welche hier geprüft wird, ist vom Typ (3) und lautet:

  'Jeder SLG besitzt einen SIB vom Typ `Start` *als Startknoten.'*

  Die Menge $LcNodes_7$ der Komponenten, für die diese Eigenschaft geprüft werden muss, kann innerhalb der Taxonomie folgendermaßen beschrieben werden:

[3] Dieser *SIB* ist in der Bibliothek *EWIS* Base definiert.

$$LcNodes_7 =_{df} (\text{Component} \cap SLG)$$

Die zugehörige lokale Eigenschaft $LcProp_7$ kann auf folgende Art und Weise formal definiert werden:

$$LcProp_7 =_{df} \forall \ \text{node} \in LcNodes_7. \ (\text{LcUtil.hasStartNodeSet(node)})$$

15.2.2 Komponenten

Das Ablaufverhalten einer Applikation wird im *ABC-SD* komponenten-basiert spezifiziert (siehe Kapitel 5). Durch die Definition und Überprüfung lokaler Eigenschaften, welche sich auf die Verwendung und Konfiguration von Komponenten bei der Entwicklung einer Applikation beziehen, können Fehler frühzeitig während der Entwicklung aufgedeckt und behoben werden.

Folgende lokalen Eigenschaften müssen in Bezug auf die Verwendung und Konfiguration von Komponenten bei der Spezifikation der Applikationslogik mit dem *ABC-SD* gelten:

- ***Verpflichtende Parameter*** einer Komponente referenzieren Variablen, die im Rahmen der Ausführung der Komponente zwingend notwendig sind. D.h. die *JAVA*-Implementierung des Komponente geht davon aus, dass die zugehörigen Variablen gesetzt sind und auf deren Werte zugegriffen werden kann. Ist dies nicht der Fall, weil der zu einer Variablen gehörige Parameter nicht gesetzt ist, kann dies zu einem Laufzeitfehler beim Ausführen des zu einer Komponente gehörigen *JAVA*-Codes führen.

  Die lokale Eigenschaft $LcProp_8$, welche hier geprüft wird, ist vom Typ (3) und lautet:

  'Verpflichtende Parameter einer Komponente müssen immer gesetzt sein.'

  Die Menge $LcNodes_8$ der Komponenten, für die diese Eigenschaft geprüft werden muss, kann innerhalb der Taxonomie folgendermaßen beschrieben werden:

  $$LcNodes_8 =_{df} \text{Component} \cap \ \text{hasRequiredParams}$$

  Die zugehörige lokale Eigenschaft $LcProp_8$ kann auf folgende Art und Weise formal definiert werden:

  $$LcProp_8 =_{df} \forall \ \text{node} \in LcNodes_8. \\ (\text{LcUtil.hasNoUnsetRequiredParams(node)})$$

- Die für eine Komponente definierten Branches erscheinen im Graphen als Beschriftung der von dieser Komponente ausgehenden Kanten. Am Ende der Abarbeitung einer Komponente – d.h. am Ende der Ausführung des zugehörigen *JAVA*-Codes – ist festgelegt, welcher Branch verfolgt werden soll, um den Konrollfluss fortzusetzen. Alle für eine Komponente definierten ***verpflichtenden Branchnamen*** müssen also den von der Komponente ausgehenden Kanten zugeordnet sein. Sonst kann es zu Laufzeitfehlern während der Ausführung der Applikation kommen, wenn nach Abarbeitung eine Komponente eine Kante im Graphen verfolgt werden soll, die nicht gesetzt ist.

  Die lokale Eigenschaft $LcProp_9$, welche hier geprüft wird, ist vom Typ (1) und lautet:

  'Verpflichtende Branches einer Komponente müssen immer gesetzt sein.'

  Die Menge $LcNodes_9$ der Komponenten, für die diese Eigenschaft geprüft werden muss, kann innerhalb der Taxonomie folgendermaßen beschrieben werden:

  $$LcNodes_9 =_{df} \text{Component}$$

  Die zugehörige lokale Eigenschaft $LcProp_9$ kann auf folgende Art und Weise formal definiert werden:

  $$LcProp_9 =_{df} \forall \;\text{node} \in LcNodes_9. \\ (\text{LcUtil.hasNoUnsetRequiredBranches(node)})$$

- Jede Komponente besitzt eine Spezifikation, in der die Namen der möglichen ausgehenden Kanten (d.h. Branches) festgelegt werden. Alle diese für eine Komponente definierten Branches sind im Graph verfügbar und können somit zur Definition des Kontrollflusses verwendet werden. Für eine Komponente muss jeder ***definierte Branch*** innerhalb ihrer ***JAVA-Implementierung*** berücksichtigt werden. Sonst kann im Graph mithilfe dieses Branches eine Variante des Kontrollflusses spezifiziert werden, die jedoch zur Laufzeit der Applikation aufgrund der fehlenden Implementierung nie auftritt.

  Die lokale Eigenschaft $LcProp_{10}$, welche hier geprüft wird, ist vom Typ (1) und lautet:

  'Alle definierten Branches einer Komponente müssen innerhalb der JAVA-Implementierung berücksichtigt werden.'

  Die Menge $LcNodes_{10}$ der Komponenten, für die diese Eigenschaft geprüft werden muss, kann innerhalb der Taxonomie folgendermaßen beschrieben werden:

$$LcNodes_{10} =_{df} \text{Component}$$

Die zugehörige lokale Eigenschaft $LcProp_{10}$ kann auf folgende Art und Weise formal definiert werden:

$$LcProp_{10} =_{df} \forall \ \text{node} \in LcNodes_{10}.$$
$$\text{(LcUtil.allSpecifiedBranchesImplemented(node))}$$

- Die Implementierung einer Komponente legt fest, welche ausgehenden Branches diese zur Laufzeit hat. Wird ein ***Branchname innerhalb der Implementierung*** einer Komponente verwendet, der nicht über die zugehörige Spezifikation der Komponente ***im Graph verfügbar*** gemacht wird, kann dies zu einem Laufzeitfehler führen. Im Graphen besteht dann nämlich keine Möglichkeit, den zu dem innerhalb der Implementierung verwendeten Branchnamen gehörigen Teil des Kontrollflusses zu spezifizieren, da der Branchname nicht als Beschriftung einer von dieser Komponente ausgehenden Kante zur Verfügung steht.

  Die lokale Eigenschaft $LcProp_{11}$, welche hier geprüft wird, ist vom Typ (1) und lautet:

  'Alle innerhalb der Implementierung einer Komponente verwendeten Branches müssen über die Spezifikation der Komponente im Graphen verfügbar gemacht werden.'

  Die Menge $LcNodes_{11}$ der Komponenten, für die diese Eigenschaft geprüft werden muss, kann innerhalb der Taxonomie folgendermaßen beschrieben werden:

  $$LcNodes_{11} =_{df} \text{Component}$$

  Die zugehörige lokale Eigenschaft $LcProp_{11}$ kann auf folgende Art und Weise formal definiert werden:

  $$LcProp_{11} =_{df} \forall \ \text{node} \in LcNodes_{11}.$$
  $$\text{(LcUtil.allImplementedBranchesSpecified(node))}$$

- Die ***Implementierung einer Komponente*** muss in *JAVA* realisiert bzw. automatisch aus der komponenten-basierten Spezifikation generiert werden.

  Die lokale Eigenschaft $LcProp_{12}$, welche hier geprüft wird, ist vom Typ (1) und lautet:

'Für jede im Rahmen der Spezifikation der Applikationslogik verwendete Komponente muss eine Implementierung zur Verfügung stehen.'

Die Menge $LcNodes_{12}$ der Komponenten, für die diese Eigenschaft geprüft werden muss, kann innerhalb der Taxonomie folgendermaßen beschrieben werden:

$$LcNodes_{12} =_{df} \text{Component}$$

Die zugehörige lokale Eigenschaft $LcProp_{12}$ kann auf folgende Art und Weise formal definiert werden:

$$LcProp_{12} =_{df} \forall \ \text{node} \in LcNodes_{12}. \ (\text{LcUtil.isImplemented(node)})$$

- Komponenten können Ein-/Ausgabe-Parameter besitzen. Diese ***Parameter und deren Typen*** müssen in der Spezifikation und der Implementierung einer Komponente übereinstimmen, damit keine Laufzeitfehler auftreten.

  Die lokale Eigenschaft $LcProp_{13}$, welche hier geprüft wird, ist vom Typ (1) und lautet:

  'Die Parameter einer Komponente sowie deren Typen müssen in der Spezifikation und der Implementierung der Komponenten übereinstimmen.'

  Die Menge $LcNodes_{13}$ der Komponenten, für die diese Eigenschaft geprüft werden muss, kann innerhalb der Taxonomie folgendermaßen beschrieben werden:

$$LcNodes_{13} =_{df} \text{Component}$$

  Die zugehörige lokale Eigenschaft $LcProp_{13}$ kann auf folgende Art und Weise formal definiert werden:

$$LcProp_{13} =_{df} \forall \ \text{node} \in LcNodes_{13}. \ (\text{LcUtil.parameterListOk(node)})$$

15.2.3 Konfiguration

Web-basierte Applikationen, die mit *ABC-SD* entwickelt werden, besitzen eine Konfigurationsdatei `web.xml` in der alle Dienste der Applikation sowie deren Konfigurationsdaten definiert sind (siehe Abschnitt 7).

Folgende lokalen Eigenschaften müssen in Bezug auf diese Konfigurationsdatei bei der Spezifikation der Koordinationsschicht einer Applikation mit *ABC-SD* gelten:

- Nur diejenigen Dienste einer web-basierten, mit dem *ABC-SD* entwickelten Applikation, die in der Konfigurationsdatei `web.xml` definiert sind, können auch ausgeführt werden. Die ***Verfügbarkeit des aktuellen Dienstes*** muss also sichergestellt werden.

  Die lokale Eigenschaft $LcProp_{14}$, welche hier geprüft wird, ist vom Typ (3) und lautet:

  'Der aktuelle Dienst muss in der Konfigurationsdatei `web.xml` *definiert sein.'*

  Die Menge $LcNodes_{14}$ der Komponenten, für die diese Eigenschaft geprüft werden muss, kann innerhalb der Taxonomie folgendermaßen beschrieben werden:

  $$LcNodes_{14} =_{df} (\text{Component} \cap SLG)$$

  Die zugehörige lokale Eigenschaft $LcProp_{14}$ kann auf folgende Art und Weise formal definiert werden:

  $$LcProp_{14} =_{df} \forall \ \text{node} \in LcNodes_{14}. \ (\text{LcUtil.currentServiceAvailable(node)})$$

- Ein *SIB* vom Typ `ServiceCall` [4] ermöglicht das Referenzieren, d.h. das Anstoßen, eines anderen Dienstes im Rahmen. Ein Dienst ist nur ausführbar, wenn er in der Konfigurationdatei `web.xml` definiert ist. Die ***Verfügbarkeit*** bei der Entwicklung der Koordinationsschicht einer Applikation ***referenzierter Dienste*** muss garantiert werden, indem sie in der Konfigurationdatei `web.xml` definiert werden. Sonst kann ein Laufzeitfehler auftreten.

  Die lokale Eigenschaft $LcProp_{15}$, welche hier geprüft wird, ist vom Typ (2) und lautet:

  'Alle referenzierten Dienste müssen in der Konfigurationsdatei `web.xml` *definiert sein.'*

  Die Menge $LcNodes_{15}$ der Komponenten, für die diese Eigenschaft geprüft werden muss, kann innerhalb der Taxonomie folgendermaßen beschrieben werden:

  $$LcNodes_{15} =_{df} (SIB \cap \texttt{ServiceCall})$$

  Die zugehörige lokale Eigenschaft $LcProp_{15}$ kann auf folgende Art und Weise formal definiert werden:

[4]Dieser *SIB* ist in der Bibliothek PwisBase definiert.

$$LcProp_{15} =_{df} \forall \text{ node} \in LcNodes_{15}.$$
$$(\text{LcUtil.referencedServiceAvailable(node)})$$

- Einige *SIB*s – z.B. `ServiceProperty2CallContext` [5] bzw. `ServiceProperty-2InitContext` [6] – greifen auf Konfigurationsdaten zu. Ein Dienst kann dabei nur auf diejenigen Konfigurationsdaten zugreifen, die für ihn in der Konfigurationdatei `web.xml` definiert sind. Die ***Verfügbarkeit benötigter Konfigurationsdaten*** muss gewährleistet werden, da sonst ein Laufzeitfehler auftreten kann.

  Die lokale Eigenschaft $LcProp_{16}$, welche hier geprüft wird, ist vom Typ (3) und lautet:

  'Alle in einem Dienst benötigten Konfigurationsdaten müssen in der `web.xml` *definiert sein.'*

  Die Menge $LcNodes_{16}$ der Komponenten, für die diese Eigenschaft geprüft werden muss, kann innerhalb der Taxonomie folgendermaßen beschrieben werden:

  $$LcNodes_{16} =_{df} \text{Component} \cap \text{usesConfigData}$$

  Die zugehörige lokale Eigenschaft $LcProp_{16}$ kann auf folgende Art und Weise formal definiert werden:

  $$LcProp_{16} =_{df} \forall \text{ node} \in LcNodes_{16}. \ (\text{LcUtil.configDataAvailable(node)})$$

15.2.4 Filesystem

Das Filesystem spielt eine wichtige Rolle während der Entwicklung und des Betriebs einer web-basierten Applikation.

Folgende lokalen Eigenschaften müssen in Bezug auf das Filesystem bei der Spezifikation der Koordinationsschicht einer Applikation mit *ABC-SD* gelten:

- Komponenten können Eingabeparameter besitzen, welche den Namen einer Datei im Filesystem angeben. Diese ***referenzierten Dateien*** müssen existieren, da sonst ein Laufzeitfehler auftreten kann.

[5]Dieser *SIB* ist in der Bibliothek *EWIS* Base definiert.
[6]Dieser *SIB* ist in der Bibliothek PwisBase definiert.

Beispiel 15.5:

Bestimmte *SIB*s – z.B. `ShowFile` [7] bzw. `DownloadFile` [8] – referenzieren Dateien, die ein Template der Präsentationsschicht beinhalten, das während der Ausführung einer Applikation verwendet wird, um z.B. Daten anzuzeigen. ⌟

Die lokale Eigenschaft $LcProp_{17}$, welche hier geprüft wird, ist vom Typ (3) und lautet:

'Alle in Komponenten als Parameter referenzierten Dateien müssen vorhanden sein.'

Die Menge $LcNodes_{17}$ der Komponenten, für die diese Eigenschaft geprüft werden muss, kann innerhalb der Taxonomie folgendermaßen beschrieben werden:

$$LcNodes_{17} =_{df} \text{Component} \cap \text{FileAccess}$$

Die zugehörige lokale Eigenschaft $LcProp_{17}$ kann auf folgende Art und Weise formal definiert werden:

$$LcProp_{17} =_{df} \forall \;\; \text{node} \in LcNodes_{17}.\ (\text{LcUtil.usedFilesExist(node)})$$

- Komponenten können auch Eingabeparameter besitzen, welche den Namen eines Verzeichnisses Datei im Filesystem angeben, auf das während der Ausführung der Komponente zugegriffen wird. Diese ***referenzierten Verzeichnisse*** müssen existieren, da sonst ein Laufzeitfehler auftreten kann.

  Die lokale Eigenschaft $LcProp_{18}$, welche hier geprüft wird, ist vom Typ (3) und lautet:

  'Alle in Komponenten als Parameter referenzierten Verzeichnisse müssen existieren.'

  Die Menge $LcNodes_{18}$ der Komponenten, für die diese Eigenschaft geprüft werden muss, kann innerhalb der Taxonomie folgendermaßen beschrieben werden:

  $$LcNodes_{18} =_{df} \text{Component} \cap \text{DirAccess}$$

  Die zugehörige lokale Eigenschaft $LcProp_{18}$ kann auf folgende Art und Weise formal definiert werden:

[7] Dieser *SIB* ist in der Bibliothek *EWIS* Base definiert.

[8] Dieser *SIB* ist in der Bibliothek PwisBase definiert.

$$LcProp_{18} =_{df} \forall \ \text{node} \in LcNodes_{18}. \ (\text{LcUtil.usedDirectoriesExist(node)})$$

- Vor dem Bereitstellen einer Applikation muss die ***Implementierung des aktuellen Dienstes*** aus der komponenten-basierten Spezifikation seines Ablaufverhaltens generiert werden.

  Die lokale Eigenschaft $LcProp_{19}$, welche hier geprüft wird, ist vom Typ (3) und lautet:

  'Die Implementierung des aktuellen Dienstes muss vorhanden sein.'

  Die Menge $LcNodes_{19}$ der Komponenten, für die diese Eigenschaft geprüft werden muss, kann innerhalb der Taxonomie folgendermaßen beschrieben werden:

  $$LcNodes_{19} =_{df} (\text{Component} \cap SLG)$$

  Die zugehörige lokale Eigenschaft $LcProp_{19}$ kann auf folgende Art und Weise formal definiert werden:

  $$LcProp_{19} =_{df} \forall \ \text{node} \in LcNodes_{19}. \\ (\text{LcUtil.currentServiceImplemented(node)})$$

- Ein *SIB* vom Typ `ServiceCall` [9] ermöglicht des Aufrufen eines anderen Dienstes. Die ***Spezifikation*** des Ablaufverhaltens als *SLG* muss für ***alle referenzierten Dienste*** vorhanden sein, da sonst auf nicht realisierte Dienste verwiesen wird.

  Die lokale Eigenschaft $LcProp_{20}$, welche hier geprüft wird, ist vom Typ (2) und lautet:

  'Alle referenzierten Dienste müssen als SLG spezifiziert sein.'

  Die Menge $LcNodes_{20}$ der Komponenten, für die diese Eigenschaft geprüft werden muss, kann innerhalb der Taxonomie folgendermaßen beschrieben werden:

  $$LcNodes_{20} =_{df} (\text{Component} \cap \texttt{ServiceCall})$$

  Die zugehörige lokale Eigenschaft $LcProp_{20}$ kann auf folgende Art und Weise formal definiert werden:

  $$LcProp_{20} =_{df} \forall \ \text{node} \in LcNodes_{20}. \\ (\text{LcUtil.referencedServiceImplemented(node)})$$

[9] Dieser *SIB* ist in der Bibliothek PwisBase definiert.

- Ein *SIB* vom Typ *ServiceCall*[10] ermöglicht des Aufrufen eines anderen Dienstes. Ein Dienst ist nur ausführbar, wenn seine Implementierung verfügbar ist. Die Implementierung eines Dienstes wird durch die automatische Codegenierung des METAFrame *Agent Building Center* (siehe Abschnitt 5.3) erzeugt. Steht beim Bereistellen der Applikation die ***Implementierung referenzierter Dienste*** nicht zur Verfügung, wird dies zu einem Laufzeitfehler führen.

  Die lokale Eigenschaft $LcProp_{21}$, welche hier geprüft wird, ist vom Typ (2) und lautet:

  'Alle referenzierten Dienste müssen implementiert sein.'

  Die Menge $LcNodes_{21}$ der Komponenten, für die diese Eigenschaft geprüft werden muss, kann innerhalb der Taxonomie folgendermaßen beschrieben werden:

  $$LcNodes_{21} =_{df} (\text{Component} \cap \texttt{ServiceCall})$$

  Die zugehörige lokale Eigenschaft $LcProp_{21}$ kann auf folgende Art und Weise formal definiert werden:

  $$LcProp_{21} =_{df} \forall \ \text{node} \in LcNodes_{21}.$$
  $$(\text{LcUtil.referencedServiceImplemented(node)})$$

[10] Dieser *SIB* ist in der Bibliothek PWISBase definiert.

Kapitel 16

Überprüfung globaler Eigenschaften

Im Rahmen der Entwicklung einer personalisierten, web-basierten Applikation mithilfe des *ABC-SD* ist die Modellierung der Koordinationsschicht dieser Applikation, welche sich aus den Koordinationsschichten der in ihr enthaltenen Dienste zusammensetzt, zentral.

Während der komponenten-basierten Erstellung der *SLG*s für die einzelnen Dienste der zu entwickelnden Applikation müssen dabei bestimmte Aspekte in Bezug auf die Verwendung und Kombination von Komponenten berücksichtigt werden.

Diese Regeln beschreiben folgende Aspekte:

- Die Reihenfolge und Zusammenarbeit von Komponenten muss bei der Modellierung des Kontollflusses berücksichtigt werden.
- Das Vorkommen bestimmter Komponenten an ausgezeichneten Stellen muss sichergestellt werden können.
- Die Berücksichtigung der Datenkontexte (siehe Kapitel 7), welche im Rahmen des *ABC-SD* zur Verfügung stehen, und ihrer Gültigkeitsbereiche spielt eine wesentliche Rolle für die Verfügbarkeit von Daten innerhalb der Koordinationsschicht.

Beschrieben werden derartige Aspekte zum Teil im Benutzerhandbuch des *ABC-SD* ([56]) bzw. in der zugehörigen technischen Dokumentation ([12, 3, 48, 42, 35]).

Auf diese Art und Weise werden Eigenschaften von Abläufen für die Entwicklung der *SLG*s einer Applikation lose spezifiziert, d.h. Empfehlungen bzw. Vorschriften für den Aufbau dieser Graphen, welche Eigenschaften für die Wohldefiniertheit eines *SLG*s bzgl. eines betrachteten Aspekts festlegen.

Der zentrale Aspekt, auf den in diesem Kapitel der Schwerpunkt gelegt wird, ist die Überprüfung von Eigenschaften, die sich auf die Gültigkeitsbereiche der im *ABC-SD* verfügbaren Datenkontexte (siehe Kapitel 7) beziehen.

Ähnlich wie in der OO-Programmierung gibt es im Rahmen des *ABC-SD* verschiedene Gültigkeitsbereiche (Scopes).

In der OO-Programmierung ([67]) wird der Programmbereich, in dem eine Variablendefinition gilt als Gültigkeitsbereich (Scope) bezeichnet. Wir unterscheiden als Scopes Klasse, darin enthaltene Methoden sowie in diesen enthaltene Blöcke. Variablen aus einem Scope sind in einem darin enthaltenen Scope gültig. Sie sind auch sichtbar, solange sie nicht überdeckt werden. Der Compiler prüft dann, ob jeder Verwendung einer Variable auch eine Deklaration und Initialisierung innerhalb des betrachteten Scopes vorausgeht. Einer Variablen kann darüberhinaus eine Lebensdauer zugesprochen werden, die mit dem Gültigkeitsbereich nicht direkt etwas zu tun hat, z.B. kann eine Variable innerhalb eines Programms mehrfach angelegt und wieder gelöscht werden.

Eben solche Eigenschaften (d.h. `DEF-USE-DEL` Beziehungen) sollen nun im Rahmen des *ABC-SD* überprüft werden. Hier sind die Gültigkeitsbereiche, welche wir zu betrachten haben, festgelegt durch die Datenkontexte (siehe Kapitel 7), welche das *ABC-SD* zur Verfügung stellt. Allerdings gilt ein anderer Sichtbarkeitsbegriff. Die Datenkontexte sind nicht geschachtelt und erlauben kein Überdecken von Variablen.

Im Rahmen dieses Kapitels wird also eine Toolunterstützung für einen sogenannten Scope-Checker im Rahmen des *ABC-SD* vorgestellt. Dabei können die Eigenschaften, welche in Form einer Bibliothek von temporallogischen Formeln vordefiniert werden, mithilfe des in das *ABC* integrierten Modelcheckers ([58, 99]) für eine beliebige, mit dem *ABC-SD* realisierte, web-basierte Applikation während der im Entwicklungsprozess enthaltenen Validierungsphasen überprüft werden (siehe Abb. 13.1).

Die Technik des Modelchecking ([15, 94, 54, 59, 81, 52, 17]) ermöglicht die automatische Verifikation von Systemen, indem für eine mathematische Struktur (das sog. Modell) entschieden wird, ob sie eine bestimmte (temporale) Bedingung erfüllt. In unserem Fall ist das Modell gegeben als *Labelled Transition System* (*LTS*). Ein *LTS* ist ein gerichteter Graph, an dessen Knoten und Kanten atomare Informationen annotiert werden können. Bedingungen werden als Formeln in einer speziellen Logik spezifiziert, z.B. im modalen Mu-Kalkül ([7, 44, 6]). Mithilfe der Formel können dann temporale Eigenschaften auf Basis der an die Knoten und Kanten annotierten atomaren Informationen formuliert werden. Eine Formel kann sich dabei nur auf Eigenschaften bzgl. der Graphstruktur beziehen. Daten werden üblicherweise nicht berücksichtigt. Sie können aber vor Beginn des Modelcheckings mittels eines

Präprozesses in atomare Informationen eines Knotens oder einer Kante codiert werden und stehen damit dann auch entsprechend beim Spezifizieren der Formeln zur Verfügung.

Für die Überprüfung der Gültigkeit temporaler Bedingungen im Modell kann der über das Modul *MCGame* in das *ABC* integrierte Modelchecker ([58, 99]) verwendet werden. Dieser benutzt als Eingabesprache den modalen Mu-Kalkül ([7, 44, 6]) mit Rückwärtsmodalität, stellt darüber hinaus aber auch die bekannten *CTL*-Operatoren ([16]) sowie eine Rückwärts-Variante dieser *CTL*-Operatoren als Makrokonstrukte der Eingabesprache zur Verfügung. Vor der eigentlichen Überprüfung einer Bedingung erfolgt eine Übersetzung derselben in den modalen μ-Kalkül mit Rückwärtsmodalität. Ist eine Bedingung nicht für alle Knoten des Modells gültig, kann der Benutzer interaktiv über ein Modelchecking-Spiel ([84, 85, 58]) die Fehlerstelle im Modell ermitteln und den Fehler beheben. Dabei bietet das System dem Benutzer eine Menge von Knoten an, von denen aus das Modelchecking-Spiel gestartet werden kann und für die das System eine Gewinnstrategie hat. Nach Auswahl des Startknotens kann der Bennutzer das Spiel starten. Interaktiv wird der Benutzer dabei immer wieder vom System aufgefordert, den Knoten zu wählen, mit dem das Spiel fortgesetzt werden soll, bis schließlich das System gewonnen hat, d.h. einen Knoten im Modell ermittelt hat, der die angegebene temporale Bedingung nicht erfüllt.

16.1 Vorgehensweise

Bei der Überprüfung globaler Eigenschaften im Rahmen der Entwicklung personalisierter, web-basierter Applikationen mithilfe des *ABC-SD* wird der in das *ABC* integrierte Modelchecker ([58, 99]) eingesetzt.

Der Modelchecker erwartet als Eingabe

- einen knoten- und kanten-annotierten, gerichteten Graphen und
- eine temporallogische Formel.

Um also für eine mithilfe des *ABC-SD* entwickelte, web-basierte Applikation bzw. einen der in ihr enthaltenen Dienste unter Einsatz des Modelcheckers die Gültigkeit einer Eigenschaft zu prüfen, muss zunächst

- die Applikation oder der Dienst in einen knoten- und kanten-annotierten, gerichteten Graphen übergeführt werden, der vom Modelchecker als Eingabe akzeptiert wird, und

- die Eigenschaft, welche geprüft werden soll, geeignet formalisiert, d.h. in Form einer temporallogischen Formel spezifiziert werden.

Die Überprüfung einer Eigenschaft für eine Applikation oder einen ihrer Dienste erfolgt mithilfe des Modelcheckings. Das zu der Applikation oder einem ihrer Dienste gehörige Modell und die der Eigenschaft entsprechende Formel dienen dabei als Eingabe für den Modelchecker.

Ausgabe des Modelcheckers ist 'ja', wenn das Modell die Formel erfüllt, und 'nein', wenn das Modell die Formel nicht erfüllt.

Im zweiten Fall – d.h. wenn das Modell die Formel nicht erfüllt – kann der Modelchecker vom Anwendungsexperten zur Fehlerdiagnose und -analyse eingesetzt werden. Hierbei hat er verschiedene Möglichkeiten:

- ***Die Menge der Fehlerknoten*** wird vom Modelchecker berechnet und kann im Modell rot markiert angezeigt werden. Auf diese Weise erhält der Anwendungsexperte im Rahmen der Validierung einen Gesamtüberblick, welche Knoten des Modells die ausgewählte Eigenschaft nicht erfüllen.

- ***Ein beliebiger Fehlerpfad*** kann durch den Modelchecker berechnet und anschließend im Modell markiert werden. So hat der Anwendungsexperte die Möglichkeit, den Bereich innerhalb des Modells zu betrachten, in welchem der Fehler auftritt.

- ***Das Starten eines interaktiven Modelchecking Spiels*** stellt eine noch spezifischere Möglichkeit der Fehlerdiagnose dar. Dabei kann der Anwendungsexperte im Rahmen des Modelchecking-Spiels Schritt für Schritt die Entstehung des Fehlers sowie die Auswertung der Teilformeln für die ausgewählte Eigenschaft nachvollziehen (wie in [58] detailliert beschrieben).

Abb. 16.1 gibt einen Überblick über die einelnen Schritte sowie die benötigten Eingaben und die gelieferten Ausgaben.

16.2 Modellgenerierung

Die Generierung des Modells für das Modelchecking erfolgt durch einen Präprozess, welcher aus mehreren Schritten besteht. Diese sind in den folgenden Abschnitten detailliert beschrieben.

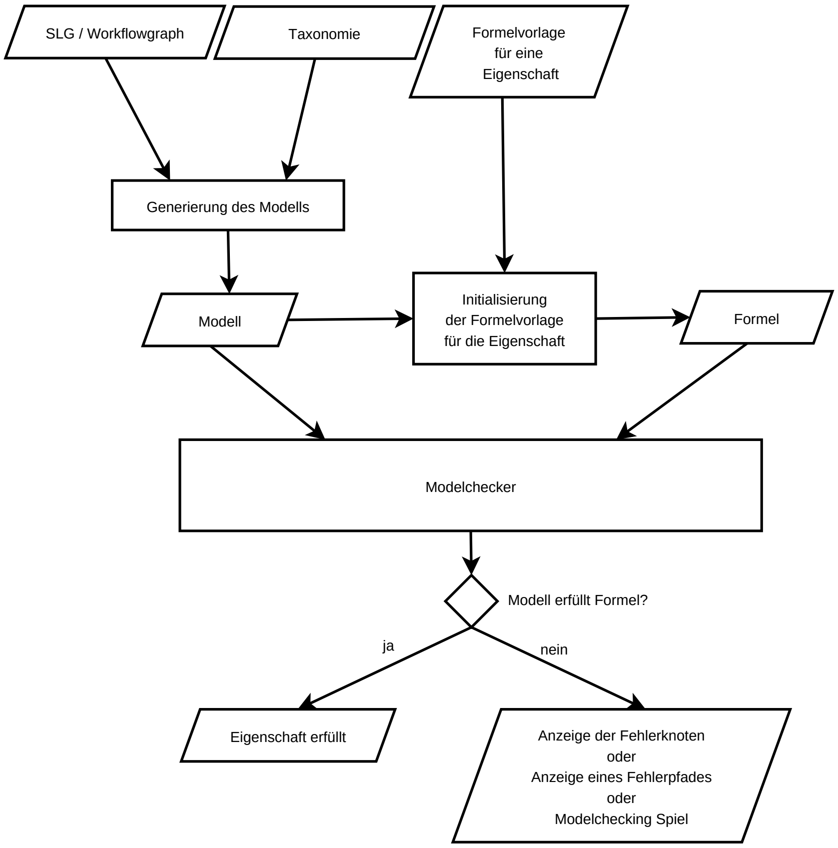


Abbildung 16.1: Einsatz des Modelcheckings zur Fehlerdiagnose

16.2.1 Initiales Modell

Zu Beginn wird das initiale Modell erzeugt, indem abhängig von der Ebene, auf welcher eine globale Eigenschaft überprüft werden soll,

- eine Kopie des Original-*SLG*s erstellt wird (dabei ergibt sich eine Struktur des Modells wie in Abb. 16.2 dargestellt)

oder

- der Workflowgraph der Applikation durch das Expandieren und Verbinden der einzelnen *SLG*s in einem Graphen dargestellt wird (dabei ergibt sich eine Struktur des Modells wie in Abb. 16.3 dargestellt).

Modell für einen Dienst

Ein *SLG* für einen Dienst hat drei wesentliche Bestandteile: Initialisierung, globale Fehlerbehandlung und die eigentliche Dienstlogik. Diese drei Bereiche sind disjunkt und nur durch den Start-Knoten miteinander verbunden. Jeder *SLG* besitzt genau einen Start-Knoten und mindestens einen End-Knoten. Abb. 16.2 skizziert ein Modell für einen Dienst.

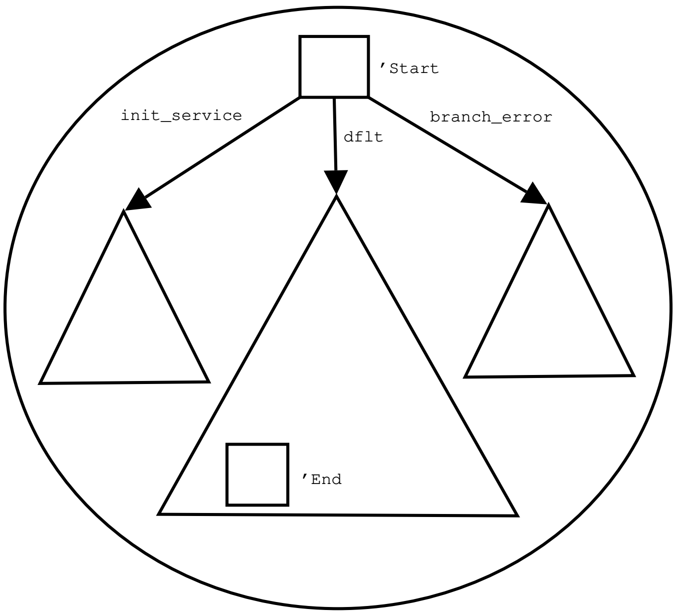


Abbildung 16.2: Modell für einen Dienst

Modell für einen Workflow

Workflows ergeben sich durch die (sinnvolle) Hintereinanderausführung mehrerer Dienste, die im Rahmen der Applikation festgelegt wird. Der entstehende Workflowgraph enthält demnach verschiedene Dienste. Dabei ist jeder Dienst als *SLG* aufgebaut. Kanten zwischen *SLG*s verlaufen immer von einem End-Knoten des einen *SLG*s zum Start-Knoten des folgenden *SLG*s.

Das berechnete Modell für einen Workflowgraphen hat dann die Struktur wie in Abb. 16.3 dargestellt.

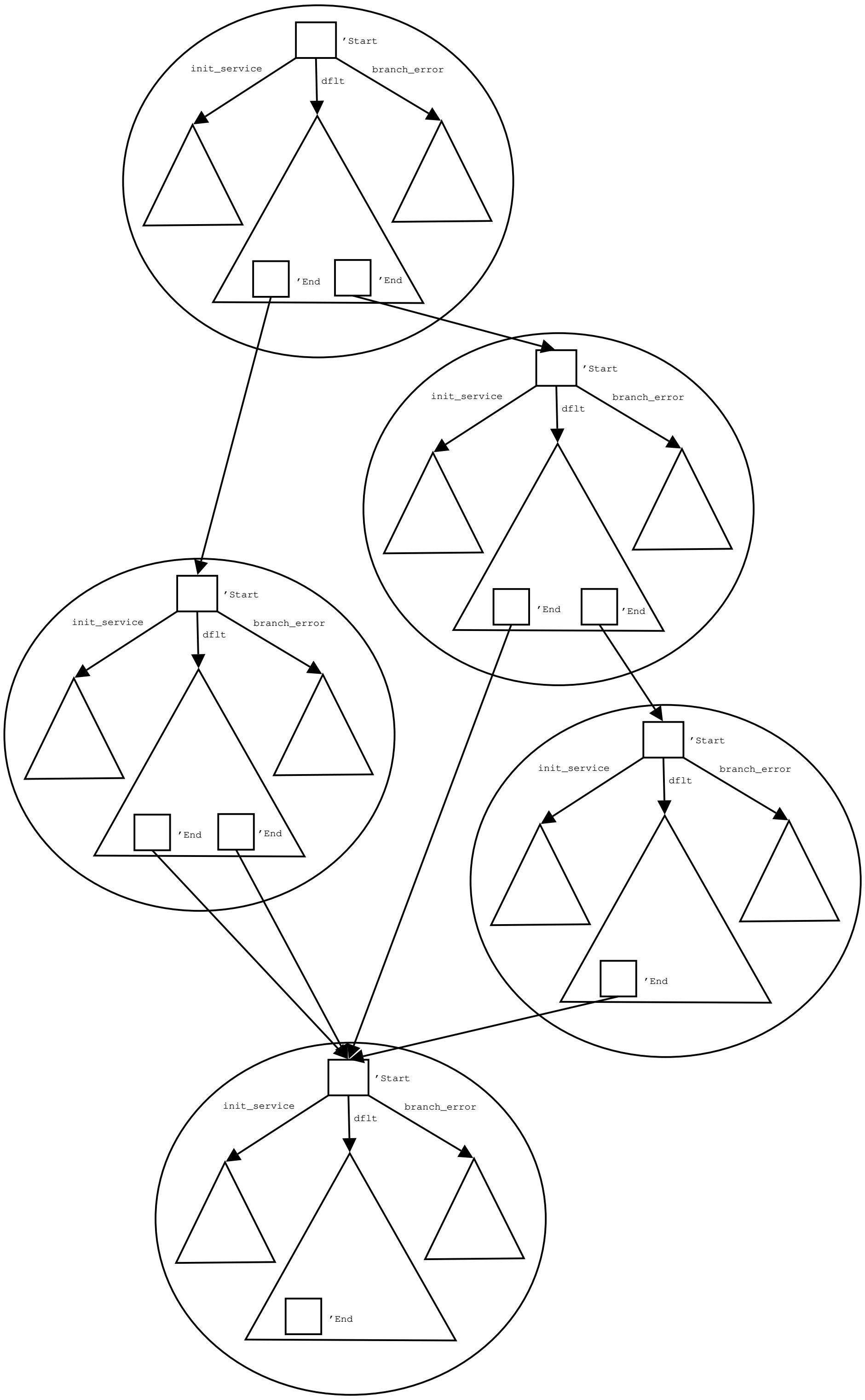


Abbildung 16.3: Modell für einen Workflow

Die einzelnen Dienste werden dabei bei der Erstellung des Workflowgraphen nach folgenden Regeln miteinander verbunden:

- Enthält ein Dienst d_1 einen `ServiceCall` *SIB* s, der einen Dienst d_2 referenziert, so wird im Workflowgraphen eine Kante einfügt, welche von diesem *SIB* s zum Startknoten des Dienstes d_2 verläuft.
- Von jedem Endknoten e eines Dienstes d ausgehend wird ebenfalls eine Kante in den Workflowgraphen eingefügt. Hierbei hängt es von dem Typ des Dienstes d ab, welchen Zielknoten diese Kante hat. Der Typ eines Dienstes ist in der Taxonomie festgelegt. Wir unterscheiden:
  - Öffentlicher Dienst: In diesem Fall verläuft die Kante vom Knoten e zum Startknoten des Dienstes *showNavbarPublic*, der die öffentliche Navigation der Applikation realisiert.
  - Privater Dienst: In diesem Fall verläuft die Kante vom Knoten e zum Startknoten des Dienstes *showNavbarPrivate*, der die personalisierte Navigation der Applikation realisiert.
  - Login-Dienst: In diesem Fall verläuft die Kante vom Knoten e zum Startknoten des Dienstes *showNavbarPrivate*, der die personalisierte Navigation der Applikation realisiert.
  - Logout-Dienst: In diesem Fall verläuft die Kante vom Knoten e zum Startknoten des Dienstes *showNavbarPublic*, der die öffentliche Navigation der Applikation realisiert.

16.2.2 Annotierung der Knoten mit Atomaren Propositionen

Im nächsten Schritt erfolgt die Annotierung der Knoten des Modells mit atomaren Propositionen[1], welche die Eigenschaften der entsprechenden Komponenten aus dem Workflowgraphen bzw. *SLG* beschreiben.

Eigenschaften von Komponenten werden bereits bei deren Entwicklung festgelegt und durch die anschließende Einordnung der Komponente in die zu einer Applikation gehörigen Taxonomie dokumentiert – wie in Kapitel 10 beschrieben.

Im Rahmen der Modellgenerierung können die Eigenschaften einer Komponente, z.B. eines *SIB*s, über die zur Applikation gehörige Taxonomie ermittelt und als atomare Proposition an den entsprechenden Knoten im Modell annotiert werden.

[1] Atomare Propositionen sind Strings, die nach bestimmten Regeln aufgebaut sind und immer mit einem ' beginnen.

Wir unterscheiden grundsätzlich zwei Arten von atomaren Propositionen:

Spezielle Eigenschaften von Komponenten

werden mittels atomarer Propositionen beschrieben und an das Modell annotiert. Sie geben an, ***was*** das Vorkommen des jeweiligen *SIB*s im Kontrollfluss der Applikation konkret bedeutet, also eine direkte Eigenschaft der Komponente, die unabhängig von einer evtl. möglichen Parametrisierung dieser Komponente gilt.

Folgende atomare Propositionen beschreiben die verschiedenen Interaktionspunkte innerhalb der Applikation:

- `'Interact` dient zur allgemeinen Kennzeichnung einer Stelle im Kontrollfluss der Applikation, an der Benutzerinteraktion stattfindet.
- `'Start` markiert den Startknoten eines Dienstes.
- `'Show` kennzeichnet Benutzerinteraktion über HTML-Seiten.
- `'ServiceEntry` markiert direkte Einsprungstellen eines Dienstes.
- `'Download` kennzeichnet Benutzerinteraktion durch Download von Daten.
- `'ServiceCall` kennzeichnet den Aufruf eines anderen Dienstes.
- `'End` kennzeichnet das Ende eines Kontrollflusses innerhalb eines Dienstes.
- `'Final` kennzeichnet einen Knoten ohne ausgehende Kanten.

*SIB*s können auch spezielle Eigenschaften bzgl. ihres Verwendungszweckes haben:

- `'InitSIB` kennzeichnet *SIB*s, die im Rahmen der Dienstinitialisierung genutzt werden.
- `'BranchErrorSIB` kennzeichnet *SIB*s, die im Rahmen der globalen Fehlerbehandlung genutzt werden.

Die Grenzen der Gültigkeitsbereiche der zu Verfügung stehenden Datenkontexte (siehe Abschnitt 6) können mittels folgender atomarer Propositionen im Modell beschrieben werden:

- `'BeginWiz` kennzeichnet den Anfang eines Wizard-Kontextes.
- `'EndWiz` kennzeichnet das Ende eines Wizard-Kontextes.

- `'CheckWiz` markiert die Stellen im Kontrollfluss, an denen die Existenz eines Wizard-Kontextes geprüft wird.
- `'SessionClear` markiert Stellen im Kontrollfluss, an denen alle Daten, die sich zu diesem Zeitpunkt im Session-Kontext befinden, gelöscht werden.

Aussagen über Variablen

werden über die der Applikation zugrunde liegende Taxonomie ermittelt und als atomare Propositionen der Form

```
'<AKTION>*<KONTEXT>*<VARNAME>
```

an das Modell annotiert. Diese enthalten Informationen über

1. die Art der Variablennutzung, d.h. die Art der ausgeführten Aktion wird spezifiziert (`<AKTION>`),
2. den Datenkontext (siehe Kapitel 7) in dem die Variable gespeichert ist (`<KONTEXT>`), und
3. den Namen der Variable (`<VARNAME>`).

Folgende Aktionen (`<AKTION>`) stehen im Rahmen der Annotierung zur Verfügung:

- `DEF` – eine Variable wird definiert
- `USE` – eine Variable wird verwendet
- `DEL` – eine Variable wird gelöscht
- `CHECK` – die Existenz einer Variable wird überprüft

Die verschiedenen Datenkontexte (`<KONTEXT>`), ihre Funktion sowie ihre Besonderheiten sind in Kapitel 7 beschrieben und werden folgendermaßen im Rahmen der Annotierung referenziert:

- `INIT` – Initialisierungs-Kontext
- `CONT` – Container-Kontext
- `SERV` – Service-Kontext
- `SESS` – Session-Kontext

- `CALL` – Call-Kontext
- `WIZ` – Wizard-Kontext
- `REQ` – Request-Kontext
- `SHOW` – Show-Kontext

Aussagen, die Variablen im Konfigurations-Kontext (siehe Abschnitt 7) betreffen, können lokal überprüft werden (siehe Abschnitt 15), daher wird dieser Datenkontext bei der Konstruktion atomarer Propositionen nicht berücksichtigt.

Der dritte Teil `<VARNAME>` der atomaren Proposition gibt den Namen der Variable an. Dieser Teil ist abhängig von der Parametrisierung einer Komponente innerhalb des *SLG*s.

16.2.3 Einfügen neuer Knoten

An bestimmten Stellen innerhalb eines *SLG*s (siehe Abb. 16.4) müssen zusätzliche Knoten eingefügt werden, die lediglich dem Zweck dienen, eine eindeutige Annotierung zu ermöglichen, und die wir deshalb als `UNIQUE` Knoten bezeichnen.

Mithilfe von Komponenten, die Benutzerinteraktion über HTML Seiten ermöglichen, können Daten – sogenannte Request-Parameter – über Links oder Buttons aus einer HTML-Seite in die Koordinationsschicht einer Applikation übermittelt werden.

Da eine HTML Seite mehrere ausgehende Links bzw. Buttons enthalten kann, kann der zugehörige `ShowFile` *SIB* entsprechend mehrere ausgehende Kanten besitzen.

Die Fragestellung, für die wir uns hier interssieren ist, welche Daten als Argumente eines Links oder Buttons in die Koordinationsschicht übermittelt werden, also welche Request-Parameter durch einen `ShowFile` *SIB* an jeder einzelnen seiner ausgehenden Kanten bereitgestellt werden.

Eine eindeutige Annotierung ist bei der Behandlung von Request-Parametern, welche im Rahmen von Benutzerinteraktion aus einer HTML-Seite in die Koordinationsschicht einer Applikation übermittelt werden, nur durch Einfügen neuer Knoten möglich, da die Request-Parameter ja den Kanten zugeordnet sind (siehe Abb. 16.4). Daher wird jeweils ein neuer `UNIQUE` Knoten zwischen einen `ShowFile` *SIB* und jeden seiner Nachfolger eingefügt. Die Information zu den über eine Kante übermittelten Request-Parametern werden dann an diese neuen Knoten annotiert, wie in Abb. 16.5 dargestellt.

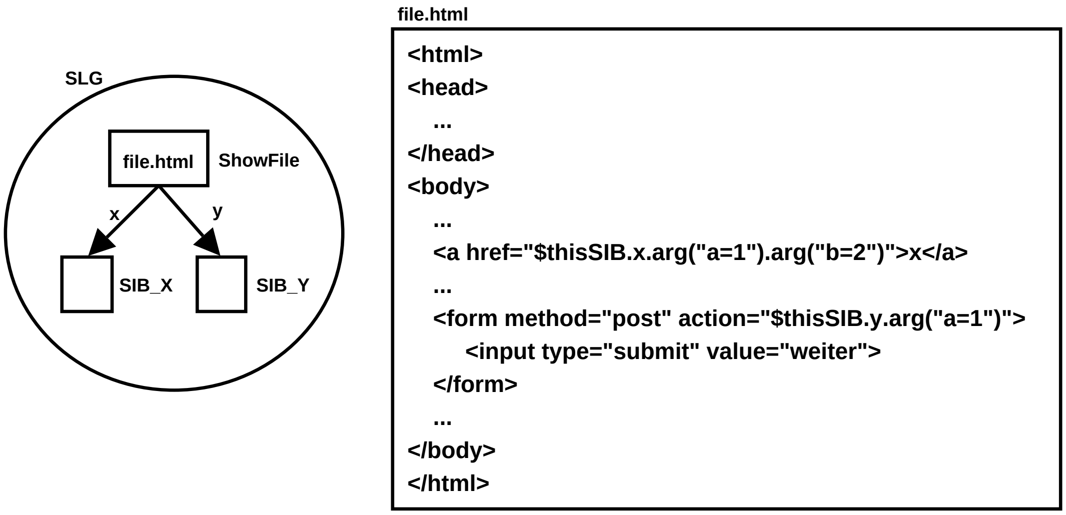


Abbildung 16.4: Behandlung von `ShowFile` *SIB*s im *SLG*

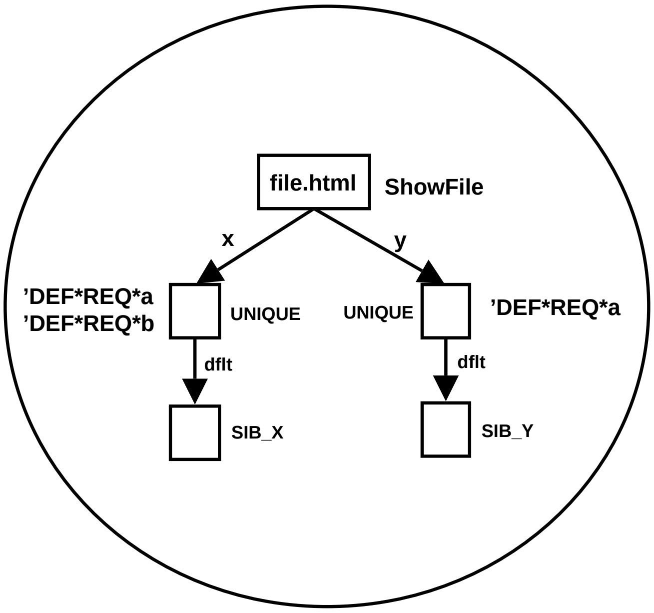


Abbildung 16.5: Behandlung von `ShowFile` *SIB*s im Modell

16.2.4 Annotierung der Kanten mit Aktionen

Die Aktionen, welche im Rahmen des Modelchecking als Annotierung der Kanten des Modells zur Verfügung stehen, werden wie üblich ([11]) aus den in der Applikation bzw. im *SLG* verwendeten Branchnamen erzeugt und in das Modell eingefügt.

16.3 Formelgenerierung

Im Rahmen der Formelgenerierung wird eine globale Eigenschaft für eine Applikation oder einen ihrer Dienste so formalisiert, dass der Modelchecker die entstehende temporallogische Formel als Eingabe akzeptiert.

16.3.1 Syntax für Temporallogische Formeln

Der in das *ABC* integrierte Modelchecker ([58, 99]) nimmt als Eingabe ein Modell, welches ein knoten- und kanten-annotierter Graph ist, sowie eine temporallogische Formel, welche mithilfe des modalen μ-Kalküls ([43, 44]) mit Vorwärts- und Rückwärtsmodalität spezifiziert wird.

Dabei bedeutet Vorwärtsmodalität, dass die Kanten des Modells in Pfeilrichtung verfolgt werden, Rückwärtsmodalität[2] entsprechend, dass die Kanten des Modells entgegen der Pfeilrichtung verfolgt werden.

Zusätzlich werden Konstrukte aus der Sprache CTL ([16]) – ebenfalls mit Vorwärts- und Rückwärtsmodalität – als Makro-Konstrukte der Eingabesprache zur Verfügung gestellt, welche intern in den modalen μ-Kalkül übersetzt werden, bevor eine Formel ausgewertet wird.

Die CTL-Operatoren werden jeweils mit Vorwärtsmodalität (dargestellt mittels des Suffixes _F) sowie mit Rückwärtsmodalität (dargestellt mittels des Suffixes _B) verwendet:

- AG (Φ) bedeutet 'Always Generally Φ'
- AF (Φ) bedeutet 'Always Finally Φ'
- EG (Φ) bedeutet 'Exists Generally Φ'
- EF (Φ) bedeutet 'Exists Finally Φ'

[2] Es handelt sich hierbei nicht um eine Rückwärtsmodalität auf dem Berechnungsbaum, sondern auf dem Modell.

- AWU (Φ_1, Φ_2) bedeutet 'Always Φ_1 Weak Until Φ_2'
- ASU (Φ_1, Φ_2) bedeutet 'Always Φ_1 Strong Until Φ_2'
- EWU (Φ_1, Φ_2) bedeutet 'Exists Φ_1 Weak Until Φ_2'
- ESU (Φ_1, Φ_2) bedeutet 'Exists Φ_1 Strong Until Φ_2'

Der Modelchecker verwendet daher für Formeln (Φ) eine Eingabesprache mit folgender Syntax:

$$
\begin{array}{rcl}
\Phi & ::= & \textit{AProp} \mid \mathrm{X} \mid \mathrm{T} \mid \mathrm{F} \mid \\
 & & (\texttt{~}\Phi\ ) \mid (\Phi\ \texttt{\&}\ \Phi) \mid (\Phi\ \texttt{|}\ \Phi) \mid (\Phi\ \texttt{=>}\ \Phi) \mid \\
 & & \textit{mu}\ \mathrm{X}.\ \Phi \mid \textit{nu}\ \mathrm{X}.\ \Phi \mid \\
 & & <\ \mathrm{Act}\ >\ \Phi \mid [\ \mathrm{Act}\ ]\ \Phi \mid\ !<\ \mathrm{Act}\ >!\ \Phi \mid\ ![\ \mathrm{Act}\ ]!\ \Phi \mid \mathrm{Macro} \\
\mathrm{Macro} & ::= & \mathrm{AG\_F}\ (\Phi) \mid \mathrm{EG\_F}\ (\Phi) \mid \mathrm{AF\_F}\ (\Phi) \mid \mathrm{EF\_F}\ (\Phi) \mid \\
 & & \mathrm{AWU\_F}\ (\Phi, \Phi) \mid \mathrm{EWU\_F}\ (\Phi, \Phi) \mid \mathrm{ASU\_F}\ (\Phi, \Phi) \mid \mathrm{ESU\_F}\ (\Phi, \Phi) \mid \\
 & & \mathrm{AG\_B}\ (\Phi) \mid \mathrm{EG\_B}\ (\Phi) \mid \mathrm{AF\_B}\ (\Phi) \mid \mathrm{EF\_B}\ (\Phi) \mid \\
 & & \mathrm{AWU\_B}\ (\Phi, \Phi) \mid \mathrm{EWU\_B}\ (\Phi, \Phi) \mid \mathrm{ASU\_B}\ (\Phi, \Phi) \mid \mathrm{ESU\_B}\ (\Phi, \Phi) \mid \\
\mathrm{Act} & ::= & .\ \mid \{\mathrm{ActList}\} \\
\mathrm{ActList} & ::= & \textit{Action} \mid \textit{Action}\ ,\ \mathrm{ActList}
\end{array}
$$

Dabei bezeichnet *AProp* eine atomare Proposition, die an einen Knoten des Modells annotiert werden kann. Eine atomare Proposition muss immer mit dem Symbol ' beginnen.

Action stellt eine Aktion dar, die an eine Kante des Modells annotiert werden kann.

Auch der Box-Operator des μ-Kalküls ist mit Vorwärtsmodalität (dargestellt mittels [ ]) sowie mit Rückwärtsmodalität (dargestellt mittels ![ ]!) verfügbar. Ebenso kann der Diamond-Operator des μ-Kalküls mit Vorwärtsmodalität (dargestellt mittels <>) sowie mit Rückwärtsmodalität (dargestellt mittels !<>!) verwendet werden.

16.3.2 Bezug zu anderen Arbeiten

Die folgenden Abschnitte geben einen kurzen Überblick über andere Arbeiten, an denen wir uns orientieren und die in dem Gebiet der Definition von Patterns in Form von temporallogischen Formeln und deren Einsatz bei der Spezifikation des Verhaltens eines Softwaresystems angesiedelt sind.

Patternsystem von Dwyer

Das Patternsystem von Dwyer ([18, 19, 20]) stellt eine Hierarchie von Patterns (siehe Abb. 16.6) zur Verfügung, welche verwendet werden können, um das Verhalten eines Systems, d.h. globale Eigenschaften desselben, zu spezifizieren.

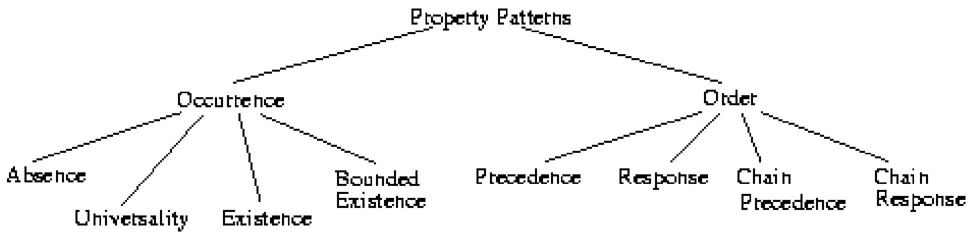


Abbildung 16.6: Pattern Hierarchie [18]

Jedes dieser Pattern betrachtet man dabei innerhalb eines bestimmten Scopes, in dem es gelten muss. Die von Dwyer behandelten Scopes für Patterns sind in Abb. 16.7 dargestellt. Ein Scope ist dabei in den Pattern-Definitionen nach [18] immer vorne geschlossen und hinten offen. Die Autoren bemerken aber auch, dass Scope-Variationen erlaubt und problemlos durchzuführen sind.

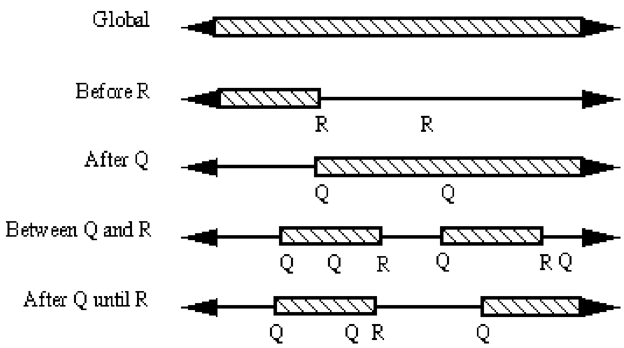


Abbildung 16.7: Geltungsbereiche für Pattern [18]

Für die Spezifikation der Eigenschaften, welche im Rahmen dieser Arbeit betrachtet werden, reicht das Patternsystem von Dwyer aus. Folgende Pattern[3] aus der Hierarchie (Abb. 16.6) werden in dieser Arbeit verwendet.

[3] In diesem Abschnitt wird die zuvor in Abschnitt 16.3.1 eingeführte Syntax zum Angeben der Formeln verwendet, um Irritationen zu vermeiden. Bei [18] wird die Negation mittels `!` und die Implikation mittels `->` dargestellt. Außerdem sind dort die temporalen Operatoren in Infix-Schreibweise verwendet.

- Absence Pattern
  - mit Scope 'Globally' ('P is false'):
    `AG ( ~P )`
  - mit Scope 'After Q' ('P is false after Q'):
    `AG ( Q => AG ( ~P ) )`
  - mit Scope 'After Q Until R' ('P is false after Q until R'):
    `AG ( Q & ~R => AWU ( ~P , R ) )`
- Universality Pattern
  - mit Scope 'Globally' ('P is true'):
    `AG ( P )`
  - mit Scope 'After Q' ('P is true after Q'):
    `AG ( Q => AG ( P ) )`
  - mit Scope 'After Q Until R' ('P is true after Q until R'):
    `AG ( Q & ~R => AWU ( P , R ) )`
- Response Pattern
  - mit Scope 'Globally' ('S responds tp P'):
    `AG ( P => AF (S))`
- Existence Pattern
  - mit Scope 'Globally' ('P becomes true'):
    `AF ( P )`

Bei Verwendung des Universality bzw. des Absence Patterns mit dem Scope 'After Q Until R' nehmen wir die Scope-Variation 'Scope vorne offen' vor:

```
AG (A => [.] AWU ( C , B ))
```

Damit drücken wir die Eigenschaft aus, dass ausgehend von einem Knoten, an dem `A` gilt, für alle Nachfolger dieses Knotens gilt, dass die Eigenschaft `C` gilt, bis wir einen Knoten mit Eigenschaft `B` erreichen.

Die Verwendung und Kombination von Vorwärts- (`_F` bzw. `[ ]` und `<>`) und Rückwärts-Modalität (`_B` bzw. `![ ]!` und `!<>!`) in den Formeln ist bei Einsatz des *MCGame*-Modelcheckers problemlos möglich.

Vorwärts-Modalität bedeutet immer 'in Pfeilrichtung der Kante', Rückwärts-Modalität immer 'entgegen der Pfeilrichtung der Kante'. Das Ganze könnte man auch direkt im Modell abbilden, indem man die Rückwärts-Kanten explizit dort aufnimmt: Wenn es im Originalgraph eine Kante `a --x--> b` gibt, dann fügt man in das Modell zusätzlich eine Kante `a <--x_B-- b` ein und kann dann Vorwärts- und Rückwärts-Kanten durch das Suffix innerhalb der Formeln unterscheiden. Diese Art der Modellgenerierung ist aber bei Verwendung des *MCGame*-Modelcheckers nicht notwendig, da dort die Unterscheidung bzgl. der Kantenrichtung von vorne herein zur Verfügung steht.

Bei Dwyer werden die meisten Pattern mit `AG` am Anfang formuliert, da hier im allgemeinen Systeme betrachtet werden, welche einen ausgezeichneten Startzustand besitzen. Bei web-basierten Applikationen bietet einerseits jeder Interaktionspunkt eine Einsprungstelle in die Applikation, auch unzusammenhängende Graphen sind so möglich und erlaubt. Andererseits überprüft der *MCGame*-Modelchecker ([58, 99]) eine gegebene Eigenschaft sowieso für alle Knoten, sodass man den äußersten Operator `AG` jeweils aus der Formel weglassen kann und dadurch sogar eine spezifischere, genauere Fehlerdiagnosemöglichkeit erhält.

Constraint Editor

Der in ([63, 65]) beschriebene Constraint Editor erlaubt einem Benutzer – i.d.R. dem Anwendungsexperten – die Instanziierung von nach Dwyer spezifizierten Constraint-Patterns mittels Füllen von Platzhaltern. Dabei kann beim Füllen der Platzhalter auf konkrete *SIB*-Typen und deren Parameter aus dem zugrundeliegenden *SLG* zugegriffen werden.

Dieser Constraint Editor bildet ein graphisches Front-End das den praktischen Einsatz der in [32] beschriebenen Ergebnisse ermöglicht, ohne dass der Benutzer sich direkt mit dem modalen μ-Kalkül befassen muss. Bei [32] erlaubt eine Erweiterung des modalen μ-Kalküls das Argumentieren über *SIB*-Typen und -Parameter sowie deren Werte innerhalb der temporallogischen Formeln.

Der Constraint-Editor soll eingesetzt werden für das Festlegen applikations-spezifischer globaler Eigenschaften durch einen Benutzer, ohne dass dieser sich direkt mit dem modalen μ-Kalkül befassen muss.

Bei den Eigenschaften bzgl. der Gültigkeitsbereiche der im *ABC-SD* zur Verfügung stehenden Datenkontexte – sog. Scoping-Regeln, die im Rahmen dieser Arbeit betrachtet werden sollen – handelt es sich um Aspekte, die für jede mit dem *ABC-SD* entwickelte, web-basierte Applikation gelten müssen. Daher wird hierfür eine Bibliothek von Scoping-Regeln erstellt, die der Anwendungsexperte mithilfe des Modelcheckers überprüfen kann, ohne sich direkt mit den einzelnen Komponenten sowie den temporallogischen Formeln auseinandersetzen zu müssen.

Ein Einsatz des Constraint Editors für die Definition von Scoping-Regeln wäre möglich, aber sehr aufwändig und unhandhabbar, wie folgendes Beispiel für eine Scoping-Regel bei Einsatz des Constraint-Editors zeigt:

Beispiel 16.1:

Betrachten wir die Aussage

'Auf ein DEF*CALL*x muss immer ein USE*CALL*x folgen'

Angenommen es gibt n *SIB*-Typen, die ein DEF*CALL*x als Eigenschaft besitzen, und m *SIB*-Typen, die ein USE*CALL*x als Eigenschaft besitzen.
Dann müsste man bei Verwendung des Constraint-Editors das Response Pattern $(n \cdot m)$-mal instanziieren, nämlich für alle möglichen Kombinationen von Vorkommen dieser *SIB*s. Dies ist sehr aufwändig und insbesondere kaum wartbar. Kommt nämlich ein neuer *SIB*-Typ hinzu, der ein USE*CALL*x als Eigenschaft besitzt, muss aus Konsistenzgründen das Response Pattern für alle n nun zusätzlich verfügbaren *SIB*-Kombinationen neu instanziiert werden.
⌟

16.3.3 Spezifikation einer Formelvorlage für eine Eigenschaft

Die Definition einer Vorlage für eine temporallogische Formel[4], welche auf einer Variable basiert, erfolgt unter Verwendung eines Platzhalters für den im Rahmen dieser Formel relevanten Variablennamen (`<<var>>`) aus dem Modell.

```
VAR String: template :=
    " ( 'USE*REQ*<<var>> => ![.]! AWU_B ( ~INTERACT , 'DEF*REQ*<<var>> ) ) ";
```

In diesem Beispiel betrachten wir als Menge der relevanten Variablen all jene, welche im Request-Kontext gespeichert sind (`REQ`) und die innerhalb des *SLG*s benutzt werden, d.h. die Aktion `USE` wird ausgeführt.

Für jede Formelvorlage müssen wir zusätzlich die Menge der relevanten Variablen spezifizieren.

Die Formeln, welche in dieser Arbeit vorgestellt werden, beschreiben Eigenschaften zur Überprüfung der Gültigkeitsbereiche der Datenkontexte (siehe Kapitel 7), welche bei der Entwicklung web-basierter Applikationen mit *ABC-SD* zur Verfügung stehen.

Das bedeutet, dass im Rahmen der hier vorgestellten Bibliothek von Scoping-Regeln i.d.R. einfache `USE-DEF-DEL`-Beziehungen unter Berücksichtigung der Gültigkeitsbereiche der bestehenden Datenkontexte geprüft werden.

[4]Die hier verwendeten Code-Beispiele sind in der Sprache HLL ([37]) angegeben.

Eine Bibliothek solcher Formelvorlagen für Scoping-Regeln im *ABC-SD* wird in Abschnitt 16.4 präsentiert.

16.3.4 Initialisierung einer Formelvorlage

Im Rahmen der Initialisierung einer Formelvorlage für ein konkretes Modell, muss die Menge der Namen von Variablen aus dem Modell ermittelt werden, welche für die ausgewählte Eigenschaft – repräsentiert durch eine Formelvorlage – relevant sind.

Hierfür gibt es eine Funktion, welche die Menge aller für die ausgewählte Eigenschaft relevanten Variablennamen für ein Modell, eine Aktion und einen Datenkontext bestimmt. Diese ruft für jeden Knoten des Modells eine entsprechende Funktion auf, welche die Menge aller für die ausgewählte Eigenschaft relevanten Variablennamen für den einzelnen Knoten bestimmt, und bildet die Vereinigung der ermittelten Teilmengen.

```
FUNCTION getVarsModel (ref PLGraph: model, String: action,
                       String: context): String Set
    VAR PLNode List: nodeList;
    VAR PLNode: next;
    VAR String Set: result;
BEGIN
    nodeList := PLGraph.getAllNodes (model);
    result := {};
    WHILE  eq ((empty (nodeList)):String, "false")  DO
        next := hd (nodeList);
        result := add (getVarsNode (next,action,context), result);
        nodeList := tl (nodeList);
    OD;
    return (result);
END


FUNCTION getVarsNode (ref PLNode: node, String: action ,
                      String: context): String Set
    VAR AProp Set: aprops; VAR AProp: nextAP;
    VAR String Set: result; VAR String: s;
    VAR String: prefix;
BEGIN
    result := {};
    aprops := McGame.getAPropSet (node);
    WHILE  eq ((empty (aprops)):String, "false")  DO
        nextAP := hd (aprops);
        prefix := "’" + action + "*" + context + "*";
        IF  String.startsWith (nextAP:String, prefix) THEN
            s := nextAP:String; String.gsub (s, prefix, "");
            result := add (s, result);
```

```
        FI;
        aprops := tl (aprops);
    OD;
    return (result);
END
```

Nach der Ermittlung der relevanten Variablennamen wird für jeden von diesen die Formelvorlage initialisiert und schließlich die Konjunktion über alle diese Teilformeln gebildet. Das Ergebnis ist eine temporallogische Formel, welche verwendet werden kann, um die ausgewählte Eigenschaft für das konkrete Modell mithilfe des Modelcheckers zu überprüfen.

```
FUNCTION initFormulaTemplate (String: template ,
                              String Set: vars) : String
    VAR String: result;
    VAR String: s;
    VAR String: nextForm;
    VAR String: nextVar;
BEGIN
    result := "";
    WHILE  eq ((empty (vars)):String, "false")  DO
        nextVar := hd (vars);
        s := template;
        nextForm := String.gsub (s, "<<var>>", nextVar);
        IF eq (result, "") THEN
            result := nextForm;
        ELSE
            result := String.add (result, " & " + nextForm);
        FI;
        vars := tl (vars);
    OD;
    print (result);
    return (result);
END
```

16.4 Katalog Globaler Temporaler Eigenschaften

Die Datenkontexte, welche im Rahmen der Entwicklung der Koordinationsschicht einer web-basierten Applikation mit *ABC-SD* zur Verfügung stehen, sind fest vorgegeben und festen Regeln unterworfen (siehe Kapitel 7).

Diese Scoping-Regeln müssen eingehalten werden, damit die entwickelte Applikation bzgl. der innerhalb der Koordinationsschicht verwendeten Variablen zuverlässig funktionieren kann.

Jeder, der mit *ABC-SD* web-basierte Applikationen entwickelt, ist selbst für die Einhaltung dieser Regeln verantwortlich.

Durch eine Formalisierung dieser Regeln sowie eine Toolunterstützung bei der Überprüfung der Regeln mittels Modelchecking, erhält der Anwendungsexperte bei der Entwicklung personalisierter, web-basierter Applikationen mit dem *ABC-SD* – d.h. beim Erstellen der *SLG*s – die Möglichkeit, von den Erfahrungen anderer zu profitieren.

„Design patterns help you to learn from others' sucesses
instead of your own failures“
(Mark Johnson[5])

Die globalen Eigenschaften, welche überprüft werden, sind insofern von großem Nutzen, weil sie so allgemein gehalten sind, dass sie für jede mit dem *ABC-SD* realisierte web-basierte Applikation angewendet werden können. Dennoch können dadurch viele der üblicherweise im Rahmen der Entwicklung auftretenden Fehler (bzgl. der allgemeinen Struktur eines *SLG*s und der Verwendung von Variablen aus den verschiedenen Datenkontexten) bereits in der Entwicklungsphase aufgedeckt und behoben werden.

„A design pattern systematically names, motivates, and explains a general design that addresses a recurring design problem in object-oriented systems. It describes the problem, the solution, when to apply the solution, and its consequences. It also gives implementation hints and examples. The solution is a general arrangement of objects and classes that solve the problem. The solution is customized and implemented to solve the problem in a particular context.“
(Erich Gamma et.al. [25])

Formeln, die sich auf den gerellen Aufbau eines *SLG*s beziehen, sind in Anhang C zusammengestellt. In den folgenden Abschnitten werden die Scoping-Regeln präsentiert und detailliert beschrieben.

16.4.1 Initialisierungs-Kontext

Beschreibung

Auf den Initialisierungs-Kontext kann nur während der Dienstinitialisierung zugegriffen werden. Diese findet im Initialisierungsteil eines Dienstes statt, d.h. dem vom

[5] zitiert von B. Eckel [21]

Startknoten aus über den init_service Branch erreichbaren Teil des *SLG*s.

Im Initialisierungs-Kontext muss für die verwendeten Variablen gelten, dass sie immer vor ihrer Verwendung initialisiert worden sind, d.h. ausgehend von der Verwendung einer beliebigen Variable <<var>> ('USE*INIT*<<var>>) muss rückwärts auf allen Pfaden immer die zugehörige Initialisierung ('DEF*INIT*<<var>>) dieser Variable zu finden sein, bevor der Startknoten ('Start) erreicht wird.

Formelvorlage

'Ausgehend von einer Variablenverwendung im Intialisierungs-Kontext darf rückwärts auf allen Pfaden nicht der Startknoten erreicht werden, bevor die zugehörige Variableninitialisierung gefunden worden ist.'

In Anlehnung an das Absence Pattern mit links offenem Scope 'After Q Until R' spezifizieren wir die Formelvorlage für diese Eigenschaft 'P is false after Q until R' folgendermaßen, wobei Q = 'USE*INIT*<<var>>, R = 'DEF*INIT*<<var>> und P = 'Start verwendet werden:

```
⋀ <<var>> ∈ getVarsModel(model,"USE","INIT")

'USE*INIT*<<var>> => ![.]! AWU_B ( ~ 'Start , 'DEF*INIT*<<var>> )
```

16.4.2 Show-Kontext

Beschreibung

Der Show-Kontext dient dazu, Daten für die Visualisierung im Rahmen einer HTML Seite o.ä. (PDF, CSV, etc.) bereitzustellen. Dabei ist der zu einem Knoten mit Benutzerinteraktion gehörige Show-Kontext innerhalb des *SLG*s ausgehend von diesem Knoten rückwärts auf allen Pfaden bis zum Erreichen des nächsten Knotens mit Benutzerinteraktion zugreifbar.

Im Show-Kontext muss für die verwendeten Variablen gelten, dass sie immer vor ihrer Verwendung initialisiert worden sind, d.h. ausgehend von der Verwendung einer beliebigen Variable <<var>> ('USE*SHOW*<<var>>) muss rückwärts auf allen Pfaden immer die zugehörige Initialisierung ('DEF*SHOW*<<var>>) dieser Variable zu finden sein, bevor der nächste Interaktionsknoten ('Interact) erreicht wird.

Formelvorlage

'Ausgehend von einer Variablenverwendung im Show-Kontext darf rückwärts auf allen Pfaden kein Interaktionsknoten erreicht werden, bevor die zugehörige Variableninitialisierung gefunden worden ist.'

In Anlehnung an das Absence Pattern mit links offenem Scope 'After Q until R' spezifizieren wir die Formelvorlage für diese Eigenschaft 'P is false after Q until R' folgendermaßen, wobei `Q = 'USE*SHOW*<<var>>`, `R = 'DEF*SHOW*<<var>>` und `P = 'Interact` verwendet werden:

```
⋀  <<var>> ∈ getVarsModel(model,"USE","SHOW")

'USE*SHOW*<<var>> =>
    ![.]! AWU_B ( ~ 'Interact , 'DEF*SHOW*<<var>> )
```

16.4.3 Call-Kontext

Beschreibung

Als Call-Kontext wird der Bereich eines *SLG*s zwischen zwei direkt aufeinander folgenden Interaktionsknoten bezeichnet.

Im Call-Kontext muss für die verwendeten Variablen gelten, dass sie immer vor ihrer Verwendung initialisiert worden sind und zwischendurch nicht wieder gelöscht werden, d.h. ausgehend von der Verwendung einer beliebigen Variable `<<var>>` (`'USE*CALL*<<var>>`) muss rückwärts auf allen Pfaden immer die zugehörige Initialisierung (`'DEF*CALL*<<var>>`) dieser Variable zu finden sein, bevor der nächste Interaktionsknoten (`'Interact`) oder ein Knoten, der diese Variable aus dem Call-Kontext entfernt (`'DEL*CALL*<<var>>`), erreicht wird.

Formelvorlage

'Ausgehend von einer Variablenverwendung im Call-Kontext darf rückwärts auf allen Pfaden kein Interaktionsknoten und kein Knoten, der diese Variable aus dem Call-Kontext entfernt, erreicht werden, bevor die zugehörige Variableninitialisierung gefunden worden ist.'

In Anlehnung an das Absence Pattern mit links offenem Scope 'After Q until R' spezifizieren wir die Formelvorlage für diese Eigenschaft 'P is false after Q until

R' folgendermaßen, wobei hier `Q = 'USE*CALL*<<var>>`, `R = 'DEF*CALL*<<var>>` und `P = ('Interact | 'DEL*CALL*<<var>>)` verwendet werden:

```
⋀ <<var>> ∈ getVarsModel(model,"USE","CALL")

'USE*CALL*<<var>> =>
     ![.]! AWU_B ( ~ ( 'Interact | 'DEL*CALL*<<var>> ) ,
               'DEF*CALL*<<var>> )
```

16.4.4 Request-Kontext

Beschreibung

Der Request-Kontext dient dazu, Daten aus einer HTML Seite innerhalb der Koordinationsschicht eines Dienstes bereitzustellen. Dabei ist der zu einem Knoten mit Benutzerinteraktion gehörige Request-Kontext innerhalb des *SLG*s ausgehend von diesem Knoten vorwärts auf allen Pfaden bis zum Erreichen des nächsten Knotens mit Benutzerinteraktion zugreifbar.

Im Request-Kontext muss für die verwendeten Variablen gelten, dass sie immer vor ihrer Verwendung initialisiert worden sind, d.h. ausgehend von der Verwendung einer beliebigen Variable `<<var>>` (`'USE*REQ*<<var>>`) muss rückwärts auf allen Pfaden immer die zugehörige Initialisierung (`'DEF*REQ*<<var>>`) dieser Variable zu finden sein, bevor der nächste Interaktionsknoten (`'Interact`) erreicht wird.

Formelvorlage

'Ausgehend von einer Variablenverwendung im Request-Kontext darf rückwärts auf allen Pfaden kein Interaktionsknoten erreicht werden, bevor die zugehörige Variableninitialisierung gefunden worden ist.'

In Anlehnung an das Absence Pattern mit links offenem Scope 'After Q until R' spezifizieren wir die Formelvorlage für diese Eigenschaft 'P is false after Q until R' folgendermaßen, wobei `Q = 'USE*REQ*<<var>>`, `R = 'DEF*REQ*<<var>>` und `P = 'Interact` verwendet werden:

```
⋀ <<var>> ∈ getVarsModel(model,"USE","REQ")

'USE*REQ*<<var>>  =>
     ![.]! AWU_B ( ~ 'Interact , 'DEF*REQ*<<var>> )
```

16.4.5 Session-Kontext

Beschreibung

Der Session-Kontext ist während der gesamten Zeit zugreifbar, in der ein Benutzer mit der web-basierten Applikation interagiert, erstreckt sich also in der Regel über mehrere Dienste.

Im Session-Kontext muss für die verwendeten Variablen gelten, dass sie immer vor ihrer Verwendung initialisiert worden sind, d.h. ausgehend von der Verwendung einer beliebigen Variable <<var>> ('USE*SESS*<<var>>) muss rückwärts auf allen Pfaden immer die zugehörige Initialisierung ('DEF*SESS*<<var>>) dieser Variable zu finden sein, bevor der Beginn des Workflows, d.h. der Startknoten des ersten Dienstes ('Start & ~!<.>! T) in der Ausführungsreihenfolge im Rahmen der stattfindenen Interaktion zwischen Benutzer und Applikation, erreicht wird oder eine Aktion passiert, welche die Variable auf die eine oder ander Weise aus dem Session-Kontext entfernt ('SessionClear | 'DEL*SESS*<<var>>).

Formelvorlage

'Ausgehend von einer Variablenverwendung im Session-Kontext darf rückwärts auf allen Pfaden kein Knoten, der diese Variable auf die eine oder andere Weise aus dem Session-Kontext entfernt, und auch nicht der initiale Startknoten erreicht werden, bevor die zugehörige Variableninitialisierung gefunden worden ist.'

In Anlehnung an das Absence Pattern mit links offenem Scope 'After Q until R' spezifizieren wir die Formelvorlage für diese Eigenschaft 'P is false after Q until R' folgendermaßen, wobei hier Q = 'USE*SESS*<<var>>, R = 'DEF*SESS*<<var>> und P = (('Start & ~!<.>! T) | 'SessionClear | 'DEL*SESS*<<var>>) verwendet werden:

```
⋀ <<var>> ∈ getVarsModel(model,"USE","SESS")

'USE*SESS*<<var>>  =>
     ![.]! AWU_B ( ~ ( ( 'Start & ~ !<.>! T ) |
                     'SessionClear |
                     'DEL*SESS*<<var>> ) ,
             'DEF*SESS*<<var>> )
```

16.4.6 Service-Kontext

Beschreibung

Der Service-Kontext eines Dienste ist innerhalb des gesamten, zu diesem Dienst gehörigen *SLG*s zugreifbar.

Im Service-Kontext muss für die verwendeten Variablen gelten, dass sie immer vor ihrer Verwendung initialisiert worden sind, d.h. ausgehend von der Verwendung einer beliebigen Variable <<var>> ('USE*SERV*<<var>>) muss rückwärts auf allen Pfaden immer die zugehörige Initialisierung ('DEF*SERV*<<var>>) dieser Variable innerhalb der Dienstlogik, d.h. bevor der Startknoten dieses Dienstes erreicht wird, oder innerhalb des Initialisierungsteils des *SLG*s zu finden sein.

Dabei darf die Variable auf dem Weg zwischen Verwendungsort und Initialisierung bzw. nach ihrer Definition innerhalb des Initialsierungsteils des aktuellen Dienstes nicht aus dem Service-Kontext entfernt werden.

Formelvorlage

'Ausgehend von einer Variablenverwendung im Service-Kontext darf rückwärts auf allen Pfaden kein Knoten erreicht werden, der diese Variable auf die eine oder andere Weise aus dem Service-Kontext entfernt, bevor die zugehörige Variableninitialisierung gefunden worden ist. Auch der Startknoten des aktuellen Dienstes darf nicht erreicht werden, bevor die zugehörige Variableninitialisierung gefunden worden ist, außer in dem vom Startknoten aus erreichbaren Initialisierungsteil wird die Variable initialisiert und danach nicht wieder gelöscht.'

In Anlehnung an das Absence Pattern mit links offenem Scope 'After Q until R' spezifizieren wir die Formelvorlage für diese Eigenschaft 'P is false after Q until R' folgendermaßen, wobei im Rahmen der Formelvorlage als Teilausdrücke Q = 'USE*SERV*<<var>> und P = ('DEL*SERV*<<var>> | 'Start) verwendet werden und das R als Disjunktion definiert ist.

Dabei beschreibt der linke Operand ('DEF*SERV*<<var>>) dieser Disjunktion die Situation, dass die Variableninitialisierung gefunden wird, bevor man den Startknoten erreicht. Im rechten Operanden wird spezifiziert, dass man den Startknoten ('Start) erreicht und (&) dort dann folgendes gilt: Man erreicht über den init_service Branch einen Knoten, von dem aus man schliesslich auf allen Pfaden die Stelle der Variableninitialisierung ('DEF*SERV*<<var>>) erreicht, und ab dieser Stelle auf allen Pfaden wird die Variable nicht wieder aus dem Service-Kontext gelöscht (AG_F (~'DEL*SERV*<<var>>)).

Der zweite Operand der Disjunktion wird hierbei unter Verwendung des Existence Patterns (mit Scope 'Globally') sowie des Universality Patterns (mit Scope 'Globally') gebildet.

$\bigwedge$ `<<var>>` $\in$ `getVarsModel(model,"USE","SERV")`

```
'USE*SERV*<<var>> =>
     ![.]! AWU_B ( ~ ( 'DEL*SERV*<<var>> |
                       'Start ),
               ( 'DEF*SERV*<<var>> |
                 ( 'Start &
                   <{init_service}>
                   AF_F ( 'DEF*SERV*<<var>> &
                          AG_F ( ~ 'DEL*SERV*<<var>> ) ) ) ) )
```

16.4.7 Container-Kontext

Beschreibung

Der Container-Kontext ist innerhalb der *SLG*s aller, zu einer Applikation gehörigen Dienste zugreifbar.

Im Container-Kontext muss für die verwendeten Variablen gelten, dass sie immer vor ihrer Verwendung initialisiert worden sind, d.h. ausgehend von der Verwendung einer beliebigen Variable `<<var>>` (`'USE*CONT*<<var>>`) muss rückwärts auf allen Pfaden immer die zugehörige Initialisierung (`'DEF*CONT*<<var>>`) dieser Variable innerhalb der Dienstlogik, oder innerhalb des Initialisierungsteils eines *SLG*s, den man auf diesem Weg passiert, zu finden sein.

Dabei darf die Variable auf dem Weg zwischen Verwendungsort und Initialisierung bzw. nach ihrer Definition innerhalb des Initialsierungsteils eines Dienstes nicht aus dem Service-Kontext entfernt werden.

Formelvorlage

'Ausgehend von einer Variablenverwendung im Container-Kontext darf rückwärts auf allen Pfaden kein Knoten erreicht werden, der diese Variable auf die eine oder andere Weise aus dem Service-Kontext entfernt, bevor die zugehörige Variableninitialisierung gefunden worden ist oder ein Startknoten eines Dienstes passiert wird, in dessen Intitialisierungsteil die Variable initialisiert und danach nicht wieder gelöscht wird.'

Der einzige Unterschied zu der in dem vorigen Abschnitt beschriebenen Eigenschaft (Service-Kontext) besteht darin, dass die Variableninitialisierung nicht nur in dem Initialisierungsteil des aktuellen Dienstes erfolgen kann, sondern generell in dem Initialisierungsteil eines beiliebigen Dienstes, welchen man auf dem Weg zurück passiert.

In Anlehnung an das Absence Pattern mit links offenem Scope 'After Q until R' spezifizieren wir die Formelvorlage für diese Eigenschaft 'P is false after Q until R' folgendermaßen, wobei im Rahmen der Formelvorlage als Teilausdrücke Q = 'USE*SERV*<<var>> und P = ('DEL*SERV*<<var>> | 'Start & ~!<.>! T) verwendet werden und das R als Disjunktion definiert ist.

Dabei beschreibt der linke Operand ('DEF*SERV*<<var>>) dieser Disjunktion die Situation, dass die Variableninitialisierung gefunden wird, bevor man den Startknoten erreicht. Im rechten Operanden wird spezifiziert, dass man den Startknoten ('Start) erreicht und (&) dort dann folgendes gilt: Man erreicht über den init_service Branch einen Knoten, von dem aus man schliesslich auf allen Pfaden die Stelle der Variableninitialisierung ('DEF*SERV*<<var>>) erreicht, und ab dieser Stelle auf allen Pfaden wird die Variable nicht wieder aus dem Service-Kontext gelöscht (AG_F (~'DEL*SERV*<<var>>)).

Der zweite Operand der Disjunktion wird hierbei unter Verwendung des Existence Patterns (mit Scope 'Globally') sowie des Universality Patterns (mit Scope 'Globally') gebildet.

```
⋀  <<var>> ∈ getVarsModel(model,"USE","CONT")

'USE*SERV*<<var>> =>
     ![.]! AWU_B ( ~ ( 'DEL*SERV*<<var>> |
                      ( 'Start & ~ !<.>! T ) ) ,
                ( 'DEF*SERV*<<var>> |
                  ( 'Start &
                    <{init_service}>
                    AF_F ( 'DEF*SERV*<<var>> &
                           AG_F ( ~ 'DEL*SERV*<<var>> ) ) ) ) )
```

16.4.8 Wizard-Kontext

Der Wizard-Kontext ist ein Scope, der vom Anwendungsexperten bei Bedarf eingesetzt werden kann, um Daten über mehrere Interaktionspunkte innerhalb der Applikation aufzusammeln bzw. zu halten. Somit muss der Anwendungsexperte bei Verwendung dieses Scopes die Grenzen desselben im Rahmen der Entwicklung der

Koordinationsschicht einer Applikation explizit festlegen, indem er entsprechende Komponenten in den jeweiligen *SLG* einbaut.

Beschreibung

Wenn eine Variable aus dem Wizard-Kontext verwendet wird, muss sichergestellt sein, dass vorher innerhalb des Wizard-Kontextes diese Variable definiert und der Wizard-Kontext nicht verlassen wird. Diese Eigenschaft wird in vier Teile aufgesplittet:

1. Auf dem Weg von einer Variablenverwendung zu deren Definition darf keine Grenze des Wizard-Kontextes erreicht oder die Variable aus diesem gelöscht werden.
2. Knoten, welche Benutzerinteraktion ermöglichen und auf dem Weg von einer Variablenverwendung zu deren Definition passiert werden, müssen den Wizard-Kontext erhalten.
3. Knoten, welche die Gültigkeit des Wizard-Kontextes oder die Existenz einer Variable in diesem prüfen, müssen sinnvoll in den Kontrollfluss eingebunden werden.
4. Ein Knoten, welcher eine Variable im Wizard-Kontext definiert, muss sich innerhalb der Grenzen eines Wizard-Kontextes befinden.

Formelvorlagen

1. Nach dem Absence Pattern mit Scope 'After Q Until R', wobei im Rahmen der Formelvorlage als Teilausdrücke Q='USE*WIZ*<<var>>, R='DEF*WIZ*<<var>> und P=('BeginWiz | 'EndWiz | 'SessionClear | 'DEL*WIZ*<<var>>) verwendet werden, lautet die Formel für diese Eigenschaft:

```
⋀  <<var>> ∈ getVarsModel(model,"USE","WIZ")

'USE*WIZ*<<var>> =>
    ![.]! AWU_B ( ~ ( 'BeginWiz | 'EndWiz |
                      'SessionClear | 'DEL*WIZ*<<var>> ) ,
              'DEF*WIZ*<<var>> )
```

2. Nach dem Universality Pattern mit Scope 'After Q Until R' werden im Rahmen der Formelvorlage Q='USE*WIZ*<<var>> und R='DEF*WIZ*<<var>> als Teilausdrücke verwendet. P stellt die Konjunktion zweier Aussagen dar, wobei

- ('Show => [.]('DEF*REQ*wiz & [.]'CheckWiz)) aussagt, dass jeder Show-*SIB* den Wizard-Kontext erhalten muss, indem er an alle seine Nachfolger den zum Wizard gehörigen Request-Parameter wiz weitergibt und die Existenz des Wizards im Anschluss überprüft wird ('CheckWiz), und
- ('Start => 'DEF*REQ*wiz) aussagt, dass jeder Start-*SIB* den Wizard-Kontext erhalten muss, indem er den zum Wizard gehörigen Request-Parameter wiz weitergibt.

```
⋀  <<var>> ∈ getVarsModel(model,"USE","WIZ")

'USE*WIZ*<<var>> =>
    ![.]! AWU_B (( 'Show => [.]('DEF*REQ*wiz & [.]'CheckWiz )) &
                 ( 'Start => 'DEF*REQ*wiz ) ,
               'DEF*WIZ*<<var>> )
```

3. Nach dem Universality Pattern mit Scope 'After Q Until R', wobei hier der Scope vorne geschlossen ist, wird die nächste Eigenschaft formuliert. Sie besagt, dass – ausgehend von einer Variablenverwendung Q='USE*WIZ*<<var>> auf dem Weg zu der zugehörigen Variablendefinition R='DEF*WIZ*<<var>> – Knoten, welche die Gültigkeit des Wizard-Kontextes ('CheckWiz) oder die Existenz der aktuell betrachteten Variable prüfen ('CHECK*WIZ*<<var>>), sinnvoll in den Kontrollfluss eingebunden werden müssen.

   Konkret bedeutet dies, dass 'CheckWiz Knoten rückwärts nur über einen ok Branch, und 'CHECK*WIZ*<<var>> Knoten rückwärts nur über einen exists Branch erreicht werden dürfen.

```
⋀  <<var>> ∈ getVarsModel(model,"USE","WIZ")

'USE*WIZ*<<var>> =>
    AWU_B ( ( ( !<ok>! 'CheckWiz ) |
              ( !<.>! ~ 'CheckWiz ) )
            &
            ( ( !<exists>! 'CHECK*WIZ*<<var>> ) |
              ( !<.>! ~ 'CHECK*WIZ*<<var>> ) ) ,
        'DEF*WIZ*<<var>> )
```

4. Die Eigenschaft 'Ein Knoten, welcher eine Variable im Wizard-Kontext definiert, muss sich innerhalb der Grenzen eines Wizard-Kontextes befinden' setzt sich aus mehreren Teilen zusammen.

   Nach dem Existence Pattern mit Scope 'Globally' spezifizieren wir, dass jeder Variablendefinition innerhalb des Wizard-Kontextes ('DEF*WIZ*<<var>>) auf jeden Fall ein Knoten vorausgehen muss, der den Beginn dieses Kontextes markiert (`AF_B ('BeginWiz)`).

Das Absence Pattern mit Scope 'After Q Until R', `Q='DEF*WIZ*<<var>>`, `R='BeginWiz` und `P=('EndWiz | 'SessionClear)` drückt aus, dass der Wizard-Kontext zwischen seinem Beginn und der betrachteten Variablendefinition nicht durch eine Komponente vorzeitig beendet wird.

Das Universality Pattern mit Scope 'After Q Until R', `Q='DEF*WIZ*<<var>>`, `R='BeginWiz` besagt, dass

- Show-*SIB*s (`('Show => [.]('DEF*REQ*wiz & [.]'CheckWiz))`) und
- Start-*SIB*s (`('Start => 'DEF*REQ*wiz)`)

auf dem Weg den Wizard-Kontext erhalten.

$\bigwedge$ `<<var>>` $\in$ `getVarsModel(model,"DEF","WIZ")`

```
'DEF*WIZ*<<var>> =>
     AF_B ( 'BeginWiz ) &
     AWU_B ( ~ ('EndWiz | 'SessionClear) &
             ( 'Show => [.]('DEF*REQ*wiz & [.]'CheckWiz) ) &
             ( 'Start => 'DEF*REQ*wiz ) ,
         'BeginWiz )
```

Teil VI

Zusammenfassung und Ausblick

Kapitel 17

Zusammenfassung und Ausblick

Der erste Prototyp von TeMPlUs wurde noch mit dem in [11] eingeführten Entwicklungsprozess als ein einzelner *SLG* realisiert. Innerhalb des Entwicklerteams traten dabei diverse Probleme auf, da beispielsweise das Testen der Applikation nur von einer Person, dem Anwendungsexperten, durchgeführt werden konnte. Mit zunehmender Menge an Features im Rahmen der Applikation, verzögerte sich daher das Bereitstellen einer neueren Version der Applikation jeweils zusehends.

Um einen höheren Grad an Arbeitsteiligkeit zu erreichen und mehr Kontrolle innerhalb des Entwicklungprozesses zu erhalten, wurden die im Rahmen dieser Arbeit präsentierten Konzepte entwickelt und im TeMPlUs Projekt angewendet.

- modularer Aufbau der Applikation als Dienstfamilie
- klar und detailliert beschriebener Entwicklungsprozess
- Personalisierungsframework
- Überprüfung lokaler Eigenschaften mittels LocalCheck
- Überprüfung globaler Eigenschaften mittels Modelchecking

Bei der Weiterentwicklung der TeMPlUs Applikation haben sich diese entwickelten Konzepte bestens bewährt.

Der klar und detailliert beschriebene Software-Entwicklungsprozess ermöglichte eine problemlose Koordination und Kommunikation innerhalb des Entwicklerteams, das sich zwischenzeitlich aus bis zu acht Personen zusammensetzte.

Durch den modularen Aufbau der TeMPlUs Applikation als Dienstfamilie wurde ein hohes Maß an Arbeitsteiligkeit ermöglicht. Dadurch ergaben sich viele Vorteile:

- Verantwortlichkeiten konnten so besser verteilt werden, wodurch die Zeit bis zum Bereitstellen der Applikation verkürzt werden konnte.
- Auch die Wartbarkeit der einzelnen Dienste verbesserte sich aufgrund geringerer Größe der Graphen.
- Dienste konnten ab nun separat validiert, generiert und getestet werden.

Die Administrationsfunktionalität des Personalisierungsframeworks ermöglicht ein komfortables Software-Konfigurationsmanagement sowie die Domänenmodellierung für die Applikation über eine Web-Schnittstelle. Somit können nun auch Personen, die keine SQL-Kenntnisse haben, diese Arbeitsaufgaben übernehmen.

Ausschlaggebend für die Entwicklung des in Kapitel 16 präsentierten Scope-Checkers auf Basis des Modelcheckings sowie für das Erstellen der Kataloge von lokalen (siehe Abschnitt 15.2) und globalen (siehe Abschnitt 16.4) Eigenschaften, war die Erfahrung, dass sehr viele Fehler im Rahmen der Entwicklung der Koordinationsschicht einer Applikation Flüchtigkeitsfehler sind.

Diese enstehen durch fehlerhafte Variablennamen oder das Vergessen wichtiger Komponenten an bestimmten Stellen. Gerade hier leisten die Validierungstools des *ABC* (der LocalChecker und der *MCGame*-Modelchecker) wertvolle Dienste. Sie ermöglichen das Aufdecken und Beheben vieler kleiner Fehler bereits zur Entwicklungszeit, wodurch unnötige Verzögerungen aufgrund von Iterationen im Entwicklungsprozess vermieden werden können.

Trotz aller genannten Vorteile bleiben noch viele offene Fragen. Für eine Weiterentwicklung der im Rahmen dieser Arbeit präsentierten Konzepte, könnte die Berücksichtigung der in den folgenden Abschnitten skizzierten Szenarien von entscheidender Bedeutung sein.

17.1 Heterogenität und Kommunikation

Im Rahmen dieser Arbeit haben wir bisher nur homogene Mengen von Diensten, die alle mit *ABC-SD* entwickelt worden sind, betrachtet. Der vorgestellte Personalisierungsframework erlaubt prinzipiell auch die Konfiguration der Gesamtfunktionalität einer Applikation aus einer heterogenen Menge von Diensten, die durch verschiedene Web-Container bereitgestellt werden können (vgl. Abschnitt 12.6).

In diesem Fall müssen neue Kommunikationsmittel Berücksichtigung finden, da hier kein Datenaustausch mehr über gemeinsame Speicherbereiche (shared memory) erfolgen kann. Auch die Realisierung der Zugriffsberechtigungsprüfung muss dann neu überdacht werden.

Im Rahmen einer Menge heterogener, dynamischer Webservices spielt die Interoperabilität der einzelnen Dienste, d.h. die Kommunikation und der Datenaustausch zwischen ihnen, eine wichtige Rolle.

Einfache Kommunikationsmittel wie URL-Encoding oder die Verwendung von Post-Methoden bei Web-Formularen sind immer möglich.

Hier muss aber z.B. auch die Kombination von Technologien unter Verwendung von UDDI[1] in Betracht gezogen werden. Dies ist eine Anwendung von SOAP[2], genauergesagt ein Verzeichnisdienst, der die zentrale Rolle in einem Umfeld von dynamischen Webservices spielen soll.

Dabei unterscheidet man in UDDI drei Arten der Information:

- White Pages: Eine Art Telefonbuch
- Yellow Pages: die elektronische Entsprechung der gelben Seiten, d.h. eine Art Branchenverzeichnis
- Green Pages: Informationen über Geschäftsmodell, Geschäftsprozess und technische Details des angebotenen Webservices

17.2 Taxonomie und Feature Interaction

Im Bereich der Telekommunikation ist Feature Interaction ([50, 49]) ein bekanntes Problem: Die Features in Telefonsystemen können direkt oder indirekt miteinander interagieren. Beispielsweise kann ein Feature ein anderes direkt aufrufen, oder ein Feature beansprucht die Ressourcen, die auch von einem anderen Feature benötigt werden. Dabei sind manche Formen der Feature Interaction wünschenswert bzw. notwendig, um ein bestimmtes Ziel zu erreichen, dagegen können unerwünschte Arten der Feature Interaction schwerwiegende Systemfehler verursachen.

Auch im Bereich web-basierter Applikationen wird es immer wichtiger, Feature Interaction für die im Rahmen dieser Applikationen bereitgestellten Dienste zu untersuchen.

Entsprechend der Definition aus dem Bereich Telekommunikation, können wir auf Ebene web-basierter Applikationen direkte Interaktion von Diensten durch die Verwendung von `ServiceCall` *SIB*s identifizieren, indirekte Interaktion über den Zugriff auf gemeinsame Bereiche innerhalb der Persistenzschicht.

Allerdings gibt es auch noch Datenkontexte, welche dienstübergreifend genutzt werden können, wie z.B. Wizard-Kontext, Session-Kontext, Service-Kontext, Container-

[1] Universal Description, Discovery and Integration.
[2] Simple Object Access Protocol.

Kontext oder applikations-spezifische Caches, und die noch genauer betrachtet werden müssen.

Rechte sind im Rahmen des hier vorgestellten Personalisierungsframeworks bisher für Benutzer bzw. Benutzergruppen als Zugriffsrechte vom Typ *execute* definiert und beziehen sich auf die Erlaubnis, bestimmte Features der Applikation auszuführen.

Dabei sind die Features auf Applikationsebene vergleichbar mit den Dienstprogrammen auf Ebene eines Betriebssystems. Ähnlich wie in Betriebssystemen Zugriffsrechte vom Typ *read* und *write* für Verzeichnisse und Dateien spezifiziert werden können, erscheint dies nun auch im Bereich der Rechtekonfiguration für personalisierte, webbasierte Applikationen sinnvoll.

Dabei sind aber nicht nur Zugriffsrechte für Verzeichnisse und Dateien interessant. Die Persistenzschicht einer Applikation enthält ja auch noch die von einer Applikation verwendete(n) Datenbank(en).

Wichtig sind demnach auch Informationen darüber, auf welche Typen von (persistenten) Geschäftsobjekten die einzelnen Features einer Applikation lesend bzw. schreibend zugreifen. Ein lesender Zugriff auf ein persistentes Geschäftsobjekt vom Typ T aus dem Anwendungsbereich der Applikation ist dann gleichbedeutend mit einer Datenbankanfrage an die zum Typ T zugehörige Datenbanktabelle $table_T$. Das Speichern eines persistenten Geschäftsobjekts (= schreibender Zugriff) vom Typ T entspricht dann einer Modifikation der zugehörigen Datenbanktabelle $table_T$. Lesender oder schreibender Zugriff kann auch konkret bzgl. einer oder mehrerer Spalten dieser Tabelle erlaubt oder verboten werden.

Eine genaue Spezifikation dieser Informationen kann innerhalb der zu einem Projekt gehörigen Taxonomie dokumentiert werden.

Beispiel 17.1:

Im Rahmen des TeMPlUs Projektes gibt die Features 'zu einer Lehrveranstaltung anmelden' (Dienst *registerForCourse*) und 'zu Gruppen einer Lehrveranstaltung anmelden' (Dienst *registerForGroups*).

Der Dienst *registerForCourse* greift lesend auf die participant Tabelle der Datenbank zu, um die Menge der Lehrveranstaltungen zu ermitteln, für die der aktuelle Benutzer noch nicht als Teilnehmer angemeldet ist. Diese Veranstaltungen werden dem Benutzer zur Auswahl angeboten. Nachdem der Benutzer b eine Lehrveranstaltung l ausgewählt hat, greift der Dienst schreibend auf die participant Tabelle zu und fügt dort einen neuen Eintrag an, der b mit l assoziiert, d.h. b wird als Teilnehmer der Lehrveranstaltung l gespeichert.

Den Dienst *registerForGroups* kann ein Benutzer nur dann ausführen, wenn es eine Lehrveranstaltung gibt, die Gruppen besitzt und für die der Benutzer als Teilnehmer registriert ist. Zunächst erfolgt also ein lesender Zugriff auf die course Tabelle der Datenbank zu, um die Menge der Lehrveranstaltungen zu ermitteln, für die es Gruppen gibt. Anschließend erfolgt ein lesender Zugriff auf die participant Tabelle, um diejenigen Lehrveranstaltungen heruaszufiltern, für die der aktuelle Benutzer als Teilnehmer angemeldet ist. Diese Lehrveranstaltungen werden dann dem Benutzer für die Anmeldung angeboten. ⌟

Werden derartige Informationen innerhalb der zu einer Applikation gehörigen Taxonomie festgehalten, sind Aussagen über die Abhängigkeit oder Unabhängigkeit bestimmter Features bzw. Dienste möglich. Wenn zwei Dienste d_1 und d_2 auf völlig unterschiedlichen Bereichen der Datenbank arbeiten, sind sie definitiv unabhängig voneinander. Das bedeutet, dass zwischen ihnen keine Wechselwirkung besteht und ihre Interaktion während der Testphase im Rahmen der Softwareentwicklung nicht berücksichtigt zu werden braucht.

Die Dienste aus dem obigen Beispiel sind voneinander abhängig, da der Dienst *registerForCourse* schreibend und der Dienst *registerForGroups* lesend auf dieselbe Datenbanktabelle $table_T$ zugreifen. Die Wechselwirkung zwischen diesen Diensten muss also während der Testphase im Rahmen der Softwareentwicklung auf jeden Fall überprüft werden.

Wenn die Informationen bzgl. lesender und schreibender Zugriffe auf die verschiedenen Datenbantabellen innerhalb der Taxonomie festgehalten werden, können diese auch im Rahmen der Validierung globaler Eigenschaften verwendet werden. Auf dem Workflowgraphen einer Applikation können dann beispielsweise Eigenschaften überprüft werden, welche sich auf Typen von Geschäftsobjekten beziehen, die von Diensten benötigt bzw. bereitgestellt werden.

Beispiel 17.2:

Der Dienst *login* initialisiert den benutzer-spezifischen Cache im Rahmen der TeMPlUs Applikation, indem er die Daten und Rollen des aktuellen Benutzers dort ablegt. Dies entspricht einer Eigenschaft `'DEF*UserCache*data` und `'DEF*UserCache*roles`, wenn man sich an der in Kapitel 16 verwendeten Notation orientiert.

Alle privaten Dienste der TeMPlUs Applikation greifen im Rahmen der Zugriffsberechtigungsprüfung lesend auf den benutzer-spezifischen Cache zu. Dies entspricht den Eigenschaften `'USE*UserCache*data`, analog zu der in Kapitel 16 verwendeten Notation.

Eine Eigenschaft, welche auf dem Workflowgraphen mittels Modelchecking geprüft werden kann, ist dann beispielsweise

'Bevor auf den benutzer-spezifischen Cache zugegriffen werden kann, muss dieser mit den entsprechenden Daten initialisiert worden sein'

```
'USE*UserCache*data => AF_B('DEF*UserCache*data)
```

⌟

Teil VII

Anhang

Anhang A

Verwendete Klassen und Interfaces

A.1 MetaFrame *EWIS* Projekt

A.1.1 Interface `User`

Dieses Interface ist im Paket `de.metaframe.actormgmt.user` definiert. Folgende Methoden werden im Rahmen dieses Interfaces deklariert:

- `java.lang.String getEmail ()`
  Liefert die EMail-Adresse des Benutzers.
- `javax.mail.internet.InternetAddress getInternetAddress ()`
  Liefert die EMail-Adresse des Benutzers als `InternetAddress` Objekt.
- `java.lang.String getLogin ()`
  Liefert den Benutzernamen.
- `java.lang.String getPassword ()`
  Liefert das Passwort des Benutzers.
- `java.lang.String getPasswordAnswer ()`
  Liefert die Antwort auf die Passwortfrage des Benutzers.
- `java.lang.String getPasswordQuestion ()`
  Liefert die Passwortfrage, die dem Benutzer von der Applikation gestellt wird, wenn er sein Passwort vergessen hat.
- `void setEmail (java.lang.String email)`
  Setzt die EMail-Adresse des Benutzers.

- `void setLogin (java.lang.String login)`
  Setzt einen neuen Benutzernamen.

- `void setPassword (java.lang.String pwd)`
  Setzt ein neues Passwort.

- `void setPasswordAnswer (java.lang.String answer)`
  Setzt eine neue Antwort auf die Passwortfrage.

- `void setPasswordQuestion (java.lang.String question)`
  Setzt eine neue Passwortfrage.

A.1.2 Interface `UserGroup`

Dieses Interface ist im Paket `de.metaframe.actormgmt.usergroup` definiert. Folgende Methoden werden im Rahmen dieses Interfaces deklariert:

- `java.lang.String getName()`
  Liefert den Namen der Benutzergruppe.

- `void setName(java.lang.String name)`
  Setzt den Namen der Benutzergruppe.

A.1.3 Interface `UserGroupMembership`

Dieses Interface ist im Paket `de.metaframe.actormgmt.usergroup` definiert. Folgende Methoden werden im Rahmen dieses Interfaces deklariert:

- `User getUser()`
  Liefert den zugehörigen Benutzer

- `UserGroup getUserGroup()`
  Liefert die zugehörige Benutzergruppe

- `UserGroupID getUserGroupID()`
  Liefert die eindeutige Identifikation der Benutzergruppe, deren Mitglied der zugehörige Benutzer ist.

- `UserID getUserID()`
  Liefert die eindeutige Identifikation des Benutzers, der Mitglied der zugehörigen Benutzergruppe ist.

A.1.4 Interface `Actor`

Dieses Interface ist im Paket `de.metaframe.actormgmt.actor` definiert. Folgende Methoden werden im Rahmen dieses Interfaces deklariert:

- `java.security.Permissions getPermissions ()`
  Liefert die Rechte des Objekts vom Typ `Actor`.

- `void grantPermission (java.security.Permission perm)`
  Weist dem Objekt vom Typ `Actor` ein weiteres Recht zu.

- `boolean hasPermission (java.security.Permission perm)`
  Überprüft, ob die Rechte des Objekts vom Typ `Actor` das über den Parameter spezifizierte Recht impliziert und gibt true bzw. false zurück.

- `void revokePermission (java.security.Permission perm)`
  Entzieht dem Objekt vom Typ `Actor` das über den Parameter spezifizierte Recht.

A.1.5 Interface `StorableEntity`

Dieses Interface ist im Paket `de.metaframe.foundation` definiert. Folgende Methoden werden im Rahmen dieses Interfaces deklariert:

- `SmartAdministrator getAdministrator ()`
  Liefert den Administrator, der Objekte dieses Typs verwaltet.

- `SmartID getID ()`
  Liefert die eindeutige Identifikation des Objekts.

- `void setID (SmartID id)`
  Setzt die eindeutige Identifikation des Objekts.

- `void storeEntity ()`
  Speichert das Objekt in die Datenbank durch Aufruf der `store` Methode des zugehörigen Administrators.

A.1.6 Interface `UserAdministrator`

Dieses Interface ist im Paket `de.metaframe.actormgmt.user` definiert. Folgende Methoden werden im Rahmen dieses Interfaces deklariert:

- `UserID authorizeUserWithEmail (java.lang.String email, java.lang.String password)`
  Überprüft, ob die Argumente eine Benutzerkennung autorisieren und liefert die zugehörige UserID, anderenfalls gibt die Methode null zurück.

- `UserID authorizeUserWithLogin (java.lang.String login, java.lang.String password)`
  Überprüft, ob die Argumente eine Benutzerkennung autorisieren und liefert die zugehörige UserID, anderenfalls gibt die Methode null zurück.

- `User createUser ()`
  Erzeugt ein neues User Objekt, generiert dabei eine neue UserID und setzt diese im User Objekt.

- `boolean existsUser (UserID userId)`
  Überprüft, ob es bereits ein User Objekt mit der angegebenen UserID gibt.

- `boolean existsUserEmail (java.lang.String email)`
  Überprüft, ob es bereits ein User Objekt mit der angegebenen EMail-Adresse gibt.

- `boolean existsUserLogin (java.lang.String login)`
  Überprüft, ob es bereits ein User Objekt mit dem angegebenen Benutzernamen gibt.

- `User[] getAllUsers ()`
  Holt alle Benutzer Objekte aus der Persistenzschicht.

- `User getUser (java.lang.String login)`
  Liefert das Benutzer Objekt, das durch den angegebenen Benutzernamen identifiziert wird.

- `User getUser (UserID userId)`
  Liefert das Benutzer Objekt, das durch die angegebene UserID identifiziert wird.

- `void removeUser (java.lang.String login)`
  Entfernt das Benutzer Objekt, das durch den angegebenen Benutzernamen identifiziert wird, aus der Persistenzschicht.

- `void removeUser (UserID userId)`
  Entfernt das Benutzer Objekt, das durch die angegebene UserID identifiziert wird, aus der Persistenzschicht.

A.1.7 Interface `ActorAdministrator`

Dieses Interface ist im Paket `de.metaframe.actormgmt.actor` definiert. Folgende Methoden werden im Rahmen dieses Interfaces deklariert:

- `Actor createActor ()`
  Erzeugt eine neues Actor Objekt.

- `Actor getActor (ActorID id)`
  Liefert das Actor Objekt, das durch die angegebene Id spezifiziert wird.

- `java.security.Permissions getPermissions (ActorID actorId)`
  Liefert ein Permissions Objekt, welches alle Rechte enthält, die das Actor Objekt zur Zeit innehat.

- `void grantPermission`
  `(ActorID actorId, java.security.Permission permission)`
  Weist das spezifizierte Recht dem Actor Objekt zu, der durch die angegbene Id spezifiziert wird.

- `boolean hasPermission`
  `(ActorID actorId, java.security.Permission permission)`
  Prüft, ob das angegebene Recht dem angegebenen Actor Objekt zugewiesen ist oder nicht.

- `void linkActors (ActorID parentId, ActorID childId)`
  Verlinkt zwei Actor Objekte in der Rechte Taxonomie.

- `void removeActor (ActorID actorId)`
  Entfernt ein Actor Objekt und die zu ihm gehörigen Kategorien aus der Rechte Taxonomie.

- `void revokePermission`
  `(ActorID actorId, java.security.Permission permission)`
  Entzieht das spezifizierte Recht dem Actor, der durch die angegbene Id spezifiziert wird.

- `void unlinkActors (ActorID parentId, ActorID childId)`
  Entfernt die Verlinkung zweier Actor Objekte in der Rechte Taxonomie.

A.1.8 Interface `SmartAdministrator`

Dieses Interface ist im Paket `de.metaframe.foundation` definiert. Folgende Methoden werden im Rahmen dieses Interfaces deklariert:

- `StorableEntity createEntity ()`
  Erzeugt ein neues persistentes Objekt, welches nur die neue Id enthält.

- `boolean existsEntity (SmartID id)`
  Überprüft, ob die Persistenzschicht eine Instanz enthält, welche durch die angegebene Id identifiziert wird.

- `boolean existsWithAttribute`
  `(java.lang.String attrName, java.lang.Object attrValue)`
  Überprüft, ob die Persistenzschicht wenigstens eine Instanz enthält, welche den angegebenen Attributwert hat.

- `void fillCollectionWithAllEntities`
  `(java.util.Collection collection)`
  Füllt das angegebene Collection Objekt mit allen Instanzen.

- `StorableEntity[] getAllEntities ()`
  Holt alle Instanzen aus der Persistenzschicht.

- `StorableEntity[] getEntitiesWithAttribute`
  `(java.lang.String attrName, java.lang.Object attrValue)`
  Holt alle Instanzen aus der Persistenzschicht, welche den angegebenen Attributwert haben.

- `StorableEntity[] getEntitiesWithAttributes`
  `(java.lang.String[] attrNames, java.lang.Object[] attrValues)`
  Holt alle Instanzen aus der Persistenzschicht, welche die angegebenen Attributwerte haben.

- `StorableEntity getEntity (SmartID id)`
  Holt eine Instanz aus der Persistenzschicht.

- `void removeEntitiesWithAttribute`
  `(java.lang.String attrName, java.lang.Object attrValue)`
  Entfernt alle Instanzen aus der Persistenzschicht, welche den angegebenen Attributwert haben.

- `void removeEntity (SmartID id)`
  Entfernt eine Instanz aus der Persistenzschicht.

- `void shutdown ()`
  Gibt alle benutzten Instanzen und speichert die persistente Information, die noch nicht gespeichert worden war.

- `void storeEntity (StorableEntity entity)`
  Speichert das angegebene Objekt in der Persistenzschicht.

A.2 PWISBase Projekt

A.2.1 Klasse FeatureEntity

Für die Klasse `FeatureEntity`, welche im Paket `de.unido.ls5.util` definiert ist, sind folgende Attribute vorgesehen:

- `id`: die eindeutige Identifikation eines Objektes vom Typ `FeatureEntity`
- `description`: eine Beschreibung der durch das Feature realisierten Funktionalität
- `featureId`: eine eindeutige Kurzbenennung für das Feature
- `featureName`: der Name des Features
- `webName`: der Name des Features, der auf den Webseiten angezeigt werden soll
- `webDescription`: die Beschreibung des Features, die auf den Webseiten angezeigt werden soll
- `serviceURL`: die URL des Features
- `visibility`: die Sichtbarkeit des Features auf den Webseiten, mögliche Werte sind `VISIBLE` und `HIDDEN`
- `type`: der Typ des Features, der die Art des zugehörigen Dienstes beschreibt, mögliche Werte sind `PUBLIC`, `LOGIN`, `PRIVATE`, `LOGOUT`

A.2.2 Klasse FeatureClass

Für die Klasse `FeatureClass`, welche im Paket `de.unido.ls5.util` definiert ist, sind folgende Attribute vorgesehen:

- `id`: die eindeutige Identifikation eines Objektes vom Typ `FeatureClass`
- `name`: der Name der Featureklasse
- `webName`: der Name der Featureklasse, der auf den Webseiten angezeigt werden soll
- `serviceURL`: die URL der Webseite für die Featureklasse
- `visibility`: die Sichtbarkeit der Featureklasse auf den Webseiten, mögliche Werte sind `VISIBLE` und `HIDDEN`

A.2.3 Klasse `FeatureEntityClassification`

Für die Klasse `FeatureEntityClassification`, welche im Paket `de.unido.ls5.util` definiert ist, sind folgende Attribute vorgesehen:

- `id`: die eindeutige Identifikation eines Objektes vom Typ `FeatureEntity-Classification`, welches die Zugehörigkeit eines Features zu einer Featureklasse repräsentiert
- `classID`: die eindeutige Identifikation der zugehörigen Featureklasse
- `entityID`: die eindeutige Identifikation des zugehörigen Features

A.2.4 Klasse `FeatureRoleAssociation`

Für die Klasse `FeatureRoleAssociation`, welche im Paket `de.unido.ls5.util` definiert ist, sind folgende Attribute vorgesehen:

- `id`: die eindeutige Identifikation eines Objektes vom Typ `FeatureRoleAssociation`, welches die Zuweisung eines Rechts an eine Benutzergruppe repräsentiert
- `roleID`: die eindeutige Identifikation der zugehörigen Benutzergruppe
- `featureID`: die eindeutige Identifikation des zugehörigen Features

A.2.5 Klasse `UserFeatureAssociation`

Für die Klasse `UserFeatureAssociation`, welche im Paket `de.unido.ls5.util` definiert ist, sind folgende Attribute vorgesehen:

- `id`: die eindeutige Identifikation eines Objektes vom Typ `UserFeatureAssociation`, welches die Zuweisung eines Rechts an einen Benutzer repräsentiert
- `userID`: die eindeutige Identifikation des zugehörigen Benutzers
- `featureID`: die eindeutige Identifikation des zugehörige Features

A.2.6 Klasse `SmartInfoAssociation`

Für die Klasse `SmartInfoAssociation`, welche im Paket `de.unido.ls5.admin` definiert ist, sind folgende Attribute vorgesehen:

- `id`: die eindeutige Identifikation eines Objektes vom Typ `SmartInfoAssociation`
- `membershipKey`: die eindeutige Identifikation des zugehörigen Objekt, welches die Mitgliedschaft eines Benutzers in einer Benutzergruppe repräsentiert
- `roleKey`: die eindeutige Identifikation der Benutzergruppe
- `infoKey`: die eindeutige Identifikation der zugehörigen applikations-spezifischen Rollendaten, die Geschäftobjekte der konkreten Applikation realisiert werden müssen, wie in Abschnitt 11.2.5 am Beispiel TEMPLUS beschrieben

A.2.7 Klasse `WisStorableEntityImplJDBC`

Diese Klasse ist im Paket `de.unido.ls5.admin` definiert und erweitert die Klasse `de.metaframe.foundation.jdbc.StorableEntityImplJDBC` um folgende Methode, die im Rahmen der Realisierung selbstgeschriebener Datenbankanfragen benötigt wird.

- `void setSmartAdmin (SmartAdministratorImplJDBC admin)`
  Setzt den SmartAdministrator für das persistente Geschäftsobjekt.

A.2.8 Interface `SmartAdministratorDependency`

Dieses Interface ist im Paket `de.unido.ls5.admin` definiert und erweitert das Interface `de.metaframe.foundation.SmartAdministrator` (siehe Anhang A.1.8) um Methoden zur Verwaltung von Administrator-Abhängigkeiten auf Smart-Ebene. Folgende Methoden werden im Rahmen dieses Interfaces deklariert:

- ```
  void addSmartAdministratorDependency
     (String key, SmartAdministratorDependency admin)
     throws DBAdministratorException
  ```
  Fügt einen SmartAdministrator zur Menge der bekannten Administratoren hinzu.
- ```
  void addUserAdministratorDependency
     (String key, UserAdministratorDependency admin)
     throws DBAdministratorException
  ```
  Fügt den UserAdministrator zur Menge der bekannten Administratoren hinzu.

- `void addUserGroupMembershipAdministratorDependency`
  `(String key, UserGroupMembershipAdministratorDependency admin)`
  `throws DBAdministratorException`
  Fügt den UserGroupMembershipAdministrator zur Menge der bekannten Administratoren hinzu.

- `SmartAdministrator getSmartAdministrator (String key)`
  `throws DBAdministratorException`
  Liefert einen Administrator anhand des spezifizierten Schlüssels.

- `JDBCDBAdministrator getJDBCAdmin ()`
  Liefert den zugehörigen JDBCAdministrator.

A.2.9 Interface `JDBCAdministratorDependency`

Dieses Interface ist im Paket `de.unido.ls5.admin` definiert und deklariert Methoden zur Verwaltung von Administrator-Abhängigkeiten auf JDBC-Ebene. Folgende Methoden werden im Rahmen dieses Interfaces deklariert:

- `void addJDBCDBAdministratorDependency`
  `(String key, JDBCDBAdministrator admin)`
  `throws DBAdministratorException`

- `JDBCDBAdministrator getJDBCDBAdministrator (String key)`
  `throws DBAdministratorException`

A.3 Das PwisUser Projekt

A.3.1 Klasse `PWISUser`

Folgende Attribute sind für die Klasse `PWISUser`, welche im Paket `de.unido.ls5.wis.user` definiert ist, zusätzlich zu denen aus ihrer Oberklasse `WisUserImplJDBC` vorgesehen:

- `lastName`: der Nachnme des Benutzers
- `firstName`: der Vorname des Benutzers
- `dateOfBirth`: das Geburtsdatum des Benutzers

- `title`: der Titel des Benutzers
- `sex`: das Geschlecht des Benutzers
- `addresses`: die Adressen des Benutzers (siehe Klasse `UserAddress`)
- `emailAddresses`: die E-Mail-Adressen des Benutzers (siehe Klasse `UserEmail`)
- `phoneNumbers`: die Telefonnummern des Benutzers (siehe Klasse `UserPhone`)
- `status`: der Status des Benutzers innerhalb der Applikation
- `dateOfRegistration`: Datum und Uhrzeit der Registrierung als Benutzer der Applikation
- `dateOfLastLogin`: Datum und Uhrzeit der letzten Benutzung der Applikation
- `numOfFailedLogins`: Anzahl der fehlgeschlagenen Login-Versuche

A.3.2 Klasse `StudentData`

Folgende Attribute sind für die Klasse `StudentData`, welche innerhalb des Pakets `de.unido.ls5.wis.user` definiert ist, vorgesehen:

- `id`: die eindeutige Identifikation eines Objektes vom Typ `StudentData`
- `userKey`: die eindeutige Identifikation des zugehörigen Objekts vom Typ `PWIS-User`
- `matrNo`: die Matrikelnummer des Studenten
- `semester`: das Semester, in dem der Student sich gerade befindet
- `studyCourse`: der Studiengang, für den der Student eingeschrieben ist

Anhang B

Die DTD für die Dokumentation der DB-Struktur

```
<!ELEMENT DBDOC (PROJECTNAME,(TABLEDOC)*)>
<!ELEMENT TABLEDOC (NAME,DESCRIPTION,CLASS,SUPERCLASS?,ADMINNAME,
                    ADMINCLASS,COMPOUND?,(COLUMN)+,(KEY)*,(FKEY)*,
                    (STATUS)*,(FLAG)*,(FILEPREFIX)?,(FILE)*)>
<!ELEMENT PROJECTNAME (#PCDATA)>
<!ELEMENT NAME (#PCDATA)>
<!ELEMENT ADMINNAME (#PCDATA)>
<!ELEMENT PACKAGENAME (#PCDATA)>
<!ELEMENT CLASSNAME (#PCDATA)>
<!ELEMENT VALUE(#PCDATA)>
<!ELEMENT EXCLUDE (#PCDATA)>
<!ELEMENT DESCRIPTION (#PCDATA)>
<!ELEMENT COMMENT (#PCDATA)>
<!ELEMENT DIR (#PCDATA)>
<!ELEMENT COLUMNNAME (#PCDATA)>
<!ATTLIST COLUMNNAME VALUE CDATA "">
<!ATTLIST COLUMNNAME EXCLUDE CDATA "">
<!ELEMENT TABLENAME (#PCDATA)> <!ATTLIST TABLENAME COLUMN CDATA "id">
<!ELEMENT CLASS (PACKAGE,CLASSNAME)>
<!ELEMENT ADMINCLASS (PACKAGE,CLASSNAME)>
<!ELEMENT PACKAGE ((PACKAGENAME)+)>
<!ELEMENT COLUMN (NAME,SQLTYPE,SUPERCLASS?,DESCRIPTION?,COMMENT?)>
<!ATTLIST COLUMN PKEY (YES|NO) "NO">
<!ATTLIST COLUMN PROPERTY (NONE | NOT_NULL) "NONE">
<!ATTLIST COLUMN FILESYSTEM (YES|NO) "NO">
<!ELEMENT SUPERCLASS (PACKAGE,CLASSNAME)>
<!ELEMENT SQLTYPE EMPTY>
<!ATTLIST SQLTYPE TYPE (BOOL|INT4|FLOAT8|CHARACTER_VARYING|TIMESTAMP) "INT4">
<!ATTLIST SQLTYPE CHARNUM CDATA "0">
<!ELEMENT KEY (COLUMNNAME+)>
<!ELEMENT FKEY ((COLUMNNAME)+,TABLENAME)>
```

```
<!ELEMENT STATUS (COLUMNNAME,DESCRIPTION,STAT+)>
<!ELEMENT STAT (VALUE,NAME,DESCRIPTION)>
<!ELEMENT FLAG (COLUMNNAME,TRUE,FALSE)>
<!ATTLIST FLAG DFLT (TRUE|FALSE) "FALSE">
<!ELEMENT TRUE (DESCRIPTION)>
<!ELEMENT FALSE (DESCRIPTION)>
<!ELEMENT FILEPREFIX (PATH)>
<!ELEMENT PATH (DIR+)>
<!ELEMENT FILE (NAME)>
<!ELEMENT COMPOUND (COMPONENT+)>
<!ELEMENT COMPONENT (TABLENAME)>
<!ATTLIST COMPONENT MAIN (YES|NO) "NO">
```

Anhang C

Bibliothek Globaler Temporallogischer Eigenschaften

C.1 Dienstaufbau

Ein Dienst besitzt – wie bereits in Abschnitt 16.2.1 beschrieben – immer einen `Start` *SIB*, welcher einen verpflichtenden Branch besitzt und zwei optionale Branches:

- Über den vom `Start` *SIB* ausgehenden `dflt` Branch wird die eigentliche Dienstlogik realisiert.
- Über den vom `Start` Knoten ausgehenden `init_service` Branch kann eine Folge von Initialisierungsaktionen spezifiziert werden, die vor der ersten Ausführung des jeweiligen Dienstes abgearbeitet werden.
- Über den vom `Start` *SIB* ausgehenden `branch_error` Branch kann eine für diesen Dienst globale Fehlerbehandlung realisiert werden, die immer dann greift, wenn zur Laufzeit an einer Stelle im *SLG* eine Kante verfolgt werden soll, die nicht existiert.

C.1.1 Dienstlogik

Folgende Eigenschaften bzgl. des Aufbaus der Dienstlogik sind zu prüfen:

- Es gibt keine *SLG*s mit 'leerer Dienstlogik', d.h. Start- und Endknoten fallen nie zusammen.

  Diese Eigenschaft wird unter Verwendung des Absence Patterns mit Scope 'Globally' ausgedrückt.

```
~ ( 'Start & 'End )
```

- Jeder *SLG* besitzt eine nichtleere Dienstlogik.

  Diese Eigenschaft wird unter Verwendung des Universality Patterns mit Scope 'Globally' ausgedrückt.

  ```
  'Start => <{dflt}> T
  ```

- Ein *SLG* muss immer einen Endknoten besitzen.

  Diese Eigenschaft wird unter Verwendung des Response Patterns ('S responds to P') mit Scope 'Globally' sowie `S='End` und `P=!<{dflt}>! 'Start` ausgedrückt.

  ```
  !<{dflt}>! 'Start => AF_F ( 'End )
  ```

- Zwischen zwei Startknoten muss ein Endknoten liegen.

  Diese Eigenschaft wird unter Verwendung des Absence Patterns und Scope 'After Q until R' ausgedrückt, wobei `Q=!<{dflt}>! 'Start`, `P='Start` und `R='End` ist.

  ```
  !<{dflt}>! 'Start => AWU_F ( ~ 'Start , 'End )
  ```

- Ein Endknoten ist immer Interaktionsknoten.

  Diese Eigenschaft wird unter Verwendung des Universality Patterns mit Scope 'Globally' ausgedrückt.

  ```
  'End => 'Interact
  ```

- Knoten, die das Ende des Kontrollflusses innerhalb eines Dienstes markieren (z.B. `ServiceCall`, `Download` oder `ShowFile` ohne ausgehende Kanten), müssen als Endknoten markiert sein.

  Diese Eigenschaft wird unter Verwendung des Universality Patterns mit Scope 'Globally' ausgedrückt.

  ```
  'Final => 'End
  ```

C.1.2 Dienstinitialisierung

Folgende Eigenschaften müssen geprüft werden:

- Bei der Initialisierung eines Dienstes dürfen ausschließlich InitSIBs verwendet werden:

  Diese Eigenschaft wird unter Verwendung des Universality Patterns mit Scope 'After Q' ausgedrückt, wobei `Q=!<{init_service}>! 'Start` und `P='InitSIB` ist.

```
!<{init_service}>! 'Start => AG_F ( 'InitSIB )
```

- Jeder InitSIB befindet sich im init_service Teilbaum des Start Knotens, d.h. einen InitSIB kann man vom Start Knoten aus nicht über einen anderen Branch als den `init_service` Branch erreichen:

  Diese Eigenschaft wird unter Verwendung des Absence Patterns und Scope 'After Q Until R' ausgedrückt, mit

  – `P='InitSIB`,
  – `R='Start` und
  – `Q=(!<{branch_error,dflt,exec_error,init_call}>!'Start)& ~'Start`.

```
( !<{branch_error,dflt,exec_error,init_call}>! 'Start ) &
  ~ 'Start )
     => AWU_F ( ~ 'InitSIB , 'Start )
```

- Die Dienstinitialisierung bildet einen in sich geschlossenen Teil eines *SLG*s, welcher außer dem `init_service` Branch keine Verbindung zum restlichen Graphen besitzt.

  Diese Eigenschaft wird unter Verwendung des Absence Patterns mit Scope 'Globally' ausgedrückt.

```
~ (EF_B (!<{init_service}>! 'Start ) &
   EF_B (!<{branch_error,dflt,exec_error,init_call}>! 'Start ))
```

C.1.3 Globale Fehlerbehandlung

Es müssen folgende Eigenschaften überprüft werden:

- Die globale Fehlerbehandlung bildet einen in sich geschlossenen Teil eines *SLG*s, welcher außer dem `branch_error` Branch keine Verbindung zum restlichen Graphen besitzt und umgekehrt.

  D.h. es darf keinen Knoten geben, welcher sowohl über den `branch_error` Branch des Start Knotens, als auch einen anderen Branch dieses Knotens erreichbar ist.

  Diese Eigenschaft wird unter Verwendung des Absence Patterns mit Scope 'Globally' ausgedrückt.

  ```
  ~ (EF_B (!<{branch_error}>! 'Start ) &
     EF_B (!<{dflt,exec_error,init_call,init_service}>! 'Start ))
  ```

- BranchError *SIB*s (wie z.B. `CheckBranchError`) dürfen nur im BranchError Teil eines *SLG*s vorkommen, nicht im restlichen Teil des Dienstes:

  Diese Eigenschaft wird unter Verwendung des Absence Patterns und Scope 'After Q' ausgedrückt, mit

  – `Q=!<{dflt,exec_error,init_call,init_service}>! 'Start` und
  – `P='BranchErrorSIB`.

  ```
  !<{dflt,exec_error,init_call,init_service}>! 'Start
       => AG_F ( ~ 'BranchErrorSIB )
  ```

Glossar

ABC-SD .... 25 ff.

Diese Abkürzung bezeichnet die MetaFrame *Agent Building Center Service Definition* Entwicklungsumgebung.

Benutzer 72

Eine Person der realen Welt, die mit einer Applikation interagiert.
siehe Abschnitt 11.1.1

CSV 119,128,178

Character Separated Values. Es handelt sich hierbei um ein Dokumentenformat. Eine CSV-Datei ist eine tabellarisch strukturierte ASCII-Text-Datei, deren Elemente (Felder) durch ein bestimmtes Trennzeichen getrennt werden. Sehr häufig wird dabei als Trennzeichen das Komma verwendet, daher wird das Kürzel CSV auch oft (nicht ganz korrekt) als *Comma Separated Values* aufgelöst.
(siehe auch [46])

DAG 59

Directed Acyclic Graph.

Dienst 5

Die technische Umsetzung eines Features wird durch Verwendung von Diensten erreicht, welche bestimmte Teilfunktionalitäten innerhalb des Anwendungbereiches einer Applikation realisieren.

Dienstfamilie 8

Gibt den strukturellen Aufbau einer web-basierten Applikation durch eine Menge von Diensten vor (siehe Abb. 2.1). Dabei wird die Gesamtfunktionalität der Applikation aufgeteilt in logisch in sich geschlossene Teilfunktionalitäten. Jede von diesen entspricht einem Dienst des Anwendungsbereichs. Zusätzlich muss noch eine Meta-Koordinationsebene spezifiziert werden, welche die Kooperationsvorschrift für die einzelnen Dienste festlegt und dadurch u.a. die Navigationsstruktur der Applikation definiert.

DTD 125,211

Document Type Definition. Die DTD ist eine Deklaration in XML-Dokumenten, die die Struktur eines solchen Dokuments festlegt.

Feature 5, **73**

Ein Feature wird durch einen oder mehrere Dienste realisiert und ist Teil eines Geschäftsprozesses des Anwendungsbereichs einer Applikation.
siehe Abschnitt 11.1.2

Featureklasse 77

Features, die überwiegend auf denselben Typen von Geschäftsobjekten bzw. Daten arbeiten, werden zu einer Featureklasse gruppiert, die einen Bereich innerhalb des Anwendungsspektrums der Applikation beschreibt. Dadurch erzielt man eine Strukturierung der Gesamtfunktionalität einer Applikation.
siehe Abschnitt 11.1.5

Geschäftsprozess 4

In der Literatur ([46]) findet sich keine einheitliche Definition des Begriffs 'Geschäftsprozess'. Im Rahmen dieser Arbeit wird daher eine eigene Definition desselben verwendet, welche folgende in der Literatur häufig beschriebene Aspekte beinhaltet:
Ein Geschäftsprozess ist eine Folge von geschäftlichen Aktivitäten (= Features), die ein bestimmtes Ergebnis anstrebt. Es wird dabei ein Wert, eine Lei-

stung oder ein Produkt für Kunden erzeugt. Ein Geschäftsprozess hat einen Beginn und ein Ende sowie klar definierte In- und Outputwerte. Geschäftsprozesse werden durch den Auftrag eines externen oder internen Kunden ausgelöst und enden mit der Übernahme eines vereinbarten Ergebnisses durch den Kunden. An der Durchführung eines Geschäftsprozesses sind i.d.R. verschiedene Akteure beteiligt, wie das folgende Beispiel eines Geschäftsprozesses aus der TEMPLUS Applikation zeigt.

Beispiel 3.1:

Im Bereich der Universität können Kunden z.B. Studierende sein. Der Geschäftsprozess *'Erfassen von Studierendendaten'* beinhaltet dabei folgende Features bzw. Dienste:

- Über das Feature 'Registrierung' (d.h. den Dienst *'registerAsStudent'*) registriert sich ein studentischer Benutzer bei der Applikation.
- Der Administrator kann mithilfe des Features 'Studentendaten prüfen' (d.h. des Dienstes *'checkStudentUserData'*) die Angaben des Studierenden auf ihre Richtigkeit prüfen.
- Über das Feature 'Benutzer authentifizieren' (d.h. den Dienst *'authenticateUser'*) wird schließlich ein Benutzeraccount authentifiziert, wodurch dem Benutzer dann alle Features zur Verfügung stehen, die den ihm zugewiesenen Rollen zugeordnet sind.

⌟

GUI 127

Graphical User Interface. Dies ist die Bezeichnung für die graphische Oberfläche, über die ein Benutzer mit einer Applikation interagiert. Im Rahmen von web-basierten Applikationen besteht die GUI größtenteils aus HTML-Seiten, die über einen Brwoser angezeigt werden.

JDBC 81

Java Database Connectivity. JDBC ist eine standardisierte Datenbankschnittstelle für Java. Diese Technologie erlaubt es, das entwickelte Benutzer- und Rollenmanagement mit einer beliebigen SQL-Datenbank zu verwenden, die einen JDBC-Treiber zur Verfügung stellt.

Komponente 59

Eine Komponente ist Teil eines Systems oder kann Teil eines Systems sein. Im Rahmen der Entwicklung web-basierter Applikationen mit *ABC-SD* verwenden wir als Komponenten *SIB*s, Makros und Dienste.

LTS 158

Labelled Transition System. Ein *LTS* ist ein gerichteter Graph, an dessen Knoten und Kanten atomare Informationen annotiert werden können.

METAFrame ***Agent Building Center Service Definition*** ... 25 ff.

METAFrame *Agent Building Center Service Definition* (kurz: *ABC-SD*) ist eine graphische Entwicklungsumgebung für die komponenten-basierte Entwicklung web-basierter Applikationen (siehe Teil II).

Modelchecking 220

Die Technik des Modelchecking ([15, 94, 54, 59, 81, 52, 17]) ermöglicht die automatische Verifikation von Systemen, indem für eine mathematische Struktur (das sog. Modell) entschieden wird, ob sie eine bestimmte (temporale) Bedingung erfüllt.

μ-Kalkül 169

PDF 119,128,178

Portable Document Format. Es handelt sich hierbei um ein Dateiformat, das von Adobe Systems entwickelt wurde. PDF-Dokumente lassen sich problemlos auf verschiedenen Softwareplattformen austauschen.
(siehe auch [46])

Recht 76

Die Erlaubnis, ein Feature einer Applikation auszuführen.
siehe Abschnitt 11.1.4

Rolle 47,117

Auf Ebene der Entwicklung einer Applikation bezeichnet der Begriff Rolle einen bestimmten Aufgabenbereich innerhalb des Entwicklungsprozesses

Rolle 75

Bei der Domänenmodellierung für eine Applikation bzw. zur Laufzeit bezeichnet der Begriff Rolle einen Aufgabenbereich (näher beschrieben durch eine Menge von Features) eines konkreten Benutzers bzw. einer bestimmten Benutzergruppe.
siehe Abschnitt 11.1.3

Servlet 30

Als Servlet bezeichnet man im Rahmen der Java 2 Platform Enterprise Edition (J2EE) ein Java-Objekt, an das ein Webserver Anfragen seiner Clients delegiert, um die Antwort an den Client zu erzeugen. Der Inhalt dieser Antwort wird dabei erst im Moment der Anfrage generiert, und ist nicht bereits statisch (etwa in Form einer HTML-Seite) für den Webserver verfügbar.
Servlets stellen damit im Rahmen der J2EE-Spezifikation das Pendant zu einem CGI-Skript oder anderen Konzepten, mit denen dynamisch Web-Inhalte erstellt werden können (PHP etc.) dar. Sie sind Instanzen von Java-Klassen, die von der durch die Spezifikation definierten Klasse javax.servlet.HttpServlet abgeleitet wurden. Diese Instanzen werden bei Bedarf von einer Laufzeitumgebung, dem sogenannten Web-Container erzeugt und von ihm aus angesprochen. Der Webcontainer seinerseits kommuniziert mit dem Webserver.

SIB 28

Service Independent Building Block.
(siehe Abschnitt 5.1)

SLG 28

Service Logic Graph. Wird innerhalb des *ABC-SD* verwendet, um einen Dienst einer Applikation mithilfe eines gerichteten Graphen zu modellieren. Dabei repräsentieren die Knoten funktionale Einheiten des Anwendungsbereiches, die Kanten beschreiben den Kontrollfluss.

SOAP 193

Simple Object Access Protocol. SOAP ist ein Protokoll, mit dessen Hilfe Daten zwischen Systemen ausgetauscht und Remote Procedure Calls durchgeführt werden können. SOAP stützt sich auf die Dienste anderer Standards, XML zur Repräsentation der Daten und Internet-Protokolle der Transport- und Anwendungsschicht (vgl. TCP/IP-Referenzmodell) zur Übertragung der Nachrichten.

SQL 81,126,134

Structured Query Language. SQL ist eine Abfragesprache für relationale Datenbanken. Sie hat eine relativ einfache Syntax, die an die englische Umgangssprache angelehnt ist, und stellt eine Reihe von Befehlen zur Definition von Datenstrukturen nach der relationalen Algebra zur Manipulation von Datenbeständen (Anfügen, Bearbeiten und Löschen von Datensätzen) und zur Abfrage von Daten zur Verfügung.

SSL 30

Secure Sockets Layer. Die Standardmethode in der Industrie, um Web-Kommunikation zu schützen. SSL bezeichnet ein von der Firma Netscape entwickeltes Übertragungsprotokoll, mit dem verschlüsselte Kommunikation mittels Tunneling möglich ist.

Taxonomie 59

Eine Taxonomie ist eine Einteilung von Dingen in Gruppen (griech.: Taxon). Allgemein wird dieser Begriff verwendet für die Bezeichnung eines Klassifikationssystems, eine Systematik oder den Vorgang des Klassifizierens. Dabei werden Klassen, Mengen oder andere Konzepte, welche durch die Einteilung von

Objekten anhand bestimmter Merkmale gewonnen werden, planmäßig hierarchisch auf der Basis bestimmter Ordnungsprinzipien angeordnet und dargestellt. In der Regel ist das Ergebnis ein DAG.

UDDI 193

Universal Description, Discovery and Integration. UDDI ist ein Begriff aus der Computertechnik und bezeichnet einen Verzeichnisdienst, der die zentrale Rolle in einem Umfeld von dynamischen Webservices spielen soll.
UDDI ist eine Anwendung von SOAP. Sie stellt mit Hilfe einer SOAP Schnittstelle einen Verzeichnisdienst bereit. Dieser Verzeichnisdienst enthält Unternehmen, ihre Daten und ihre Services. Dabei kann man in UDDI zwischen drei Arten der Informationen unterscheiden: Den 'White Pages', einer Art Telefonbuch, den 'Yellow Pages', also die elektronische Entsprechung der gelben Seiten, und den 'Green Pages'. Die genaue Aufteilung mit samt den Daten, die den einzelnen Teilen entspringen werden, sind in folgender Liste ausgeführt:

- White Pages: Namensregister, Auflistung der Anbieter, Kontaktinformationen
- Yellow Pages: Branchenverzeichnis, spezifische Suche gemäß verschiedener Taxonomien, verweist auf White Pages
- Green Pages: Informationen über Geschäftsmodell, Technische Details zu den angebotenen Web Services, Auskunft über Geschäftsprozesse

UML ....

Unified Modeling Language. Graphische Sprache für die Visualisierung, Spezifikation, Konstruktion und Dokumentation von großen Software-Produkten.

URL ....

Uniform Resource Locator. Ein Uniform Resource Locator (URL engl.: einheitlicher Ortsangeber für Ressourcen) ist ein Uniform Resource Identifier (URI), der eine Ressource über ihren primären Zugriffsmechanismus, d.h. dem Ort (engl. location) der Ressource im Internet, identifiziert.

Web-basierte Applikation 5

Eine web-basierte Applikation besteht aus einer Menge von Diensten und einer Meta-Koordinationsebene, welche die Zusammenarbeit der einzelnen Dienste regelt.

Workflow 31

Ein Workflow ist eine zusammenhängende Folge von Aktionen, welche in der festgelegten Reihenfolge ausgeführt werden, wenn ein Benutzer mit einer web-basierten Applikation durch Eingabe von Daten und/oder Mausklicks interagiert. Im Rahmen eines Workflows sind Beginn und Ende, Interaktions- und Entscheidungspunkte sowie Alternativen in Ausgabepfaden klar gekennzeichnet.

WWW ....

World Wide Web.

XML ....

Extensible Markup Language. Es handelt sich hierbei um einen offenen Standard des World Wide Web Consortium (W3C) zur Erstellung strukturierter, maschinen- und menschenlesbarer Dateien. XML definiert dabei den grundsätzlichen Aufbau solcher Dateien udn wurde entwickelt als Datenformat für den Dokumentenaustausch über das Web.
(siehe auch [46])

Literaturverzeichnis

[1] ARNOLD, KEN, JAMES GOSLING und DAVID HOLMES: *The Java programming language.* The Java series ... from the source. Addison-Wesley, 3 Auflage, 2000.

[2] BEECK, M. VON DER, V. BRAUN, A. CLASSEN, A. DANNECKER, C. FRIEDRICH, D. KOSCHÜTZKI, T. MARGARIA, F. SCHREIBER und B. STEFFEN: *Graphs in METAFrame: The Unifying Power of Polymorphism.* In: *Proc. of the Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97), Enschede (NL)*, Band 1217 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 112–129, Heidelberg, Germany, March 1997. Springer-Verlag.

[3] BICKEL, MARK und BEN LINDNER: *Database HOWTO.* METAFrame Technologies GmbH, Dortmund, 2001.

[4] BOOCH, G., J. RUMBAUGH und I. JACOBSON: *The Unified Modeling Language User Guide.* Addison-Wesley, 1998.

[5] BOOCH, G., J. RUMBAUGH und I. JACOBSON: *The Unified Software Development Process.* Addison-Wesley, 1999.

[6] BRADFIELD, JULIAN: *On the expressivity on the modal mu-calculus.* In: PUECH, CLAUDE und RÜDIGER REISCHUK (Herausgeber): *STACS 96, 13th Annual Symposium on Theoretical Aspects of Computer Science, Grenoble, France, February 22-24, 1996, Proceedings*, Band 1046. Springer, 1996.

[7] BRADFIELD, JULIAN und COLIN STIRLING: *Handbook of Process Algebra*, Kapitel Modal logics and mu-calculi: An Introduction, Seiten 293–330. Elsevier, 2001.

[8] BRAUN, V., J. KREILEDER, T. MARGARIA und B. STEFFEN: *The ETI Online Service in Action.* In: *Proc. TACAS'99, Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems. Held as Part of the Joint*

European Conferences on Theory and Practice of Software (ETAPS'99), Amsterdam, The Netherlands, Band 1579 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 439–443, Heidelberg, Germany, March 22-26 1999. Springer-Verlag.

[9] Braun, V., T. Margaria und B. Steffen: *Personalized Electronic Commerce Services.* In: *Proc. ICTS'2000, 8th Int. Conference on Telecommunication Systems Modeling and Analysis, March 9-12, 2000, Nashville, Tennessee, USA*, 2000.

[10] Braun, V., T. Margaria, B. Steffen und H. Yoo: *Automatic Error Location for IN Service Definition.* In: *"Services and Visualization: Towards User-Friendly Design", also Proc. of the 2nd Int. Workshop on Advanced Intelligent Networks (AIN'97), Cesena (Italy), 4.-5. Juli 1997*, Band 1385 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 222–237, Heidelberg, Germany, March 1998. Springer-Verlag.

[11] Braun, Volker: *A Coarse-granular Approach to Software Development allowing Non-Programmers to Build and Deploy Reliable, Web-based Applications.* Dissertation, Universität Dortmund, Dortmund, Deutschland, 2001. http://eldorado.uni-dortmund.de:8080/FB4/ls5/forschung/2002/Braun.

[12] Braun, Volker, Andreas Holzmann, Susanne Bollin, Kai Plociennik und Markus Nagelmann: *The Agent Building Center: Bringing a Web Application into Operation.* MetaFrame Technologies GmbH, Dortmund, Germany, 2001.

[13] *CAS Campus ist ein flexibles System zur effizienten Veranstaltungsplanung sowie der Erstellung des Internet- und printbasierten Vorlesungsverzeichnisses.* http://www.cas.de/Produkte/Campus.

[14] Classen, Andreas: *Component Integration into* MetaFrame. Doktorarbeit, Universität Passau, 1997.

[15] Clarke, E. M., O. Grumberg und D. A. Peled: *Model Checking.* MIT Press, 1999. http://mitpress.mit.edu.

[16] Clarke, Edmund M. und E. Allen Emerson: *Design and synthesis of synchronization skeletons using branching-time temporal logic.* In: Kozen, Dexter (Herausgeber): *Logics of Programs*, Band 131 der Reihe *Lecture Notes in Computer Science*, Seiten 52–71. Springer, 1981.

[17] Cleaveland, R.: *Pragmatics of Model Checking: An STTT Special Section.* International Journal on Software Tools for Technology Tranfer, Seiten 208–218, 1999.

[18] DWYER, M., G. AVRUNIN und J. CORBETT: *A System of Specification Patterns*, 1997. http://www.cis.ksu.edu/santos/spec-patterns.

[19] DWYER, M., G. AVRUNIN und J. CORBETT: *Property Specification Patterns for Finite-State Verification.* In: *Proc. of the 2nd Workshop on Formal Methods in Software Practice*, Seiten 7–15. ACM Press, 1998.

[20] DWYER, M., G. AVRUNIN und J. CORBETT: *Patterns in Property Specifications for Finite-State Verification.* In: *Proc. of the Int. Conference on Software Engineering*, Seiten 411–420. ACM Press, 1999.

[21] ECKEL, B.: *Thinking in Patterns with Java, Revision 0.6*, 2001. http://jamesthornton.com/eckel/TIPatterns/html/Chapter01.html.

[22] FARLEY, JIM, WILLIAM CRAWFORD und DAVID FLANAGAN: *JAVA in a Nutshell.* O'Reilly, 2003.

[23] *E-Learning Plattform an der Fachhochschule Furtwangen.* http://www.felix.fh-furtwangen.de oder http://elearning.fh-furtwangen.de.

[24] FOWLER, M. und K. SCOTT: *UML Distilled – A Brief Guide to the Standard Object Modeling Language.* Addison Wesley, Second Edition Auflage, 1999.

[25] GAMMA, ERICH, RICHARD HELM, RALPH JOHNSON und JOHN VLISSIDES: *Design Patterns: Elements of Reusable Object-Oriented Software.* Professional Computing Series. Addison-Wesley.

[26] GRIFFEL, F.: *Componentware - Konzepte und Techniken eines Softwareparadigmas.* dpunkt-Verlag, 1998.

[27] GSOTTBERGER, C. und B. STEFFEN: *Endbericht für das WIS-Teilprojekt 'Koordination des Online-Angebots des Fachbereichs Informatik'*, September 2004.

[28] GSOTTBERGER, CLAUDIA: *'TEMPLUS - TEaching Management PLatform for UnivesitieS' in Kolloquium Programmiersprachen und Grundlagen der Programmierung, pp. 187–192.* Technischer Bericht AIB-2001-11, RWTH Aachen, Fachbereich Informatik, Dezember 2001. Klaus Indermark und Thomas Noll (Hrsg.).

[29] *TA Scheduler: Automatically assigns to TAs to courses.* http://www.utdallas.edu/ gupta Used at UTD since fall 2002 to make assignment for 70+ TAs to about 120 courses. Now has a web interface and a front end GUI. King, Gupta, Guballa, Mallya. 2003. An AI system developed in Prolog/CLP.

[30] *NADA: An automatic system for checking undergraduate graduation requirements.* http://www.utdallas.edu/ gupta An automatic system for checking undergraduate graduation requirements and providing advise automatically to students (built for NMSU). Implemented using Prolog + Java. Jose-Mendez, Karshmer, and Gupta. 1999.

[31] *Eine Web-Anwendung für Lehre, Studium und Forschung.* http://www.his.de/Abt1/HISLSF.

[32] HOFMANN, J.: *Program Dependent Abstract Interpretation.* Diplomarbeit, Universität Passau, 1997.

[33] HOLZMANN, ANDREAS: *Der* METAFrame *Interpreter: Entwicklung und Implementierung eines dynamischen Modulkonzeptes.* Diplomarbeit, Universität Passau, 1997.

[34] HOLZMANN, ANDREAS: *Developing with the PLGraph Library.* METAFrame Technologies GmbH, Dortmund, 2001.

[35] HOLZMANN, ANDREAS: *EWIS Survey Howto.* METAFrame Technologies GmbH, Dortmund, 2001.

[36] HOLZMANN, ANDREAS: *The Graph Tracer Module.* METAFrame Technologies GmbH, Dortmund, 2001.

[37] HOLZMANN, ANDREAS: *The High-Level-Language: Programming language of the* METAFrame *interpreter.* METAFrame Technologies GmbH, Dortmund, 2001.

[38] HOLZMANN, ANDREAS: *The SD Localcheck Module.* METAFrame Technologies GmbH, Dortmund, 2001.

[39] HOLZMANN, ANDREAS: *The SD Tracer Module.* METAFrame Technologies GmbH, Dortmund, 2001.

[40] HOLZMANN, ANDREAS: *Writing adapter specification.* METAFrame Technologies GmbH, Dortmund, 2001.

[41] HUNGAR, HARDI, TIZIANA MARGARIA und BERNHARD STEFFEN: *Test-Based Model Generation for Legacy Systems.* In: *IEEE International Test Conference (ITC)*, Charlotte, NC, September 2003.

[42] KARUSSEIT, MARTIN: *EWIS Comm HOWTO.* METAFrame Technologies GmbH, Dortmund, 2001.

[43] Kozen, Dexter: *Results on the propositional mu-calculus.* In: *International Colloquium on Automata, Languages and Programming (ICALP '82)*, Band 140 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 348–359. Springer-Verlag, Heidelberg, 1982.

[44] Kozen, Dexter: *Results on the propositional mu-calculus.* Theoretical Computer Science, 27:333–354, December 1983.

[45] Lee, David und Mihalis Yannakakis: *Principles and Methods of Testing Finite State Machines — a Survey.* Proc. IEEE, 84(8):1090–1126, 1996.

[46] *WIKIPEDIA – Die freie Enzyklopädie.* http://de.wikipedia.org.

[47] Lindner, B., T. Margaria und B. Steffen: *Ein personalisierter Internetdienst für wissenschaftliche Begutachtungsprozesse.* In: *GI-VOI-BITKOM-OCG-TeleTrusT Konferenz "Elektronische Geschäftsprozesse" (eBusiness Processes), 24-25 September, Universität Klagenfurt (Austria)*, 2001. http://syssec.uni-klu.ac.at/EBP2001/.

[48] Lindner, Ben: *EWIS User HOWTO.* MetaFrame Technologies GmbH, Dortmund, 2001.

[49] Margaria, T.: *On Modelling Feature Interactions in Telecommunication Systems.* In: *Nordic Workshop on Programming Theory, Uppsala (S), October 6-8, 1999*, 1999.

[50] Margaria, T.: *On Modelling Feature Interactions in Telecommunications.* In: *Dagstuhl Seminar on Temporal Logics for Distributed Systems - Paradigms and Algorithms, E. Clarke (Pittsburgh), U. Goltz (Hildesheim), P. Niebert (VERIMAG Grenoble), W. Penczek (Warszawa), 10.-15.10.1999*, Band Report No. 254, 2001.

[51] Margaria, T. und V. Braun: *Formal Methods and Customized Visualization: A Fruitful Symbiosis.* In: *in "Services and Visualization: Towards User-Friendly Design", Proc. of Int. Worksh. on Visual Issues for Formal Methods (VISUAL'98), Satellite Workshop of TACAS'98/ETAPS'98, Lisbon, Portugal*, Band 1385 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 190–207, Heidelberg, Germany, March 30 - April 3 1998. Springer-Verlag.

[52] Margaria, T., V. Braun und B. Steffen: *Coarse Granular Model Checking in Practice.* In: *Proc. SPIN Workshop 2001, satellite to ICSE 2001, May 2001, Toronto (Canada)*, Band 2017 der Reihe *Lecture Notes in Computer Science (LNCS)*, Heidelberg, Germany, 2001. Springer-Verlag.

[53] *Multimedia and E-Learning Services.* http://www.id.unizh.ch/mels.

[54] MERZ, STEPHAN: *Model Checking: A Tutorial Overview.* In: AL., F. CASSEZ ET (Herausgeber): *Modeling and Verification of Parallel Processes*, Band 2067 der Reihe *Lecture Notes in Computer Science*, Seiten 3–38. Springer-Verlag, Berlin, 2001.

[55] METAFrame TECHNOLOGIES GMBH, DORTMUND: *The Agent Building Center: Integrator's Guide to Conceptual SIB Design*, 2001.

[56] METAFrame TECHNOLOGIES GMBH, DORTMUND: *The Agent Building Center: User Guide*, 2001.

[57] MÜLLER-OLM, M.: *Tutorial on Model Checking and Program Analysis at the MOVEP'02 summer school in Nantes*, 2002. http://ls5-www.cs.uni-dortmund.de/ mmo/slides/movep02color.pdf.

[58] MÜLLER-OLM, M. und H. YOO: *MetaGame: An animation tool for model-checking games.* In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04), March 29-April 2 2004, Barcelona, Spain*, LNCS. Springer Verlag, 2004.

[59] MÜLLER-OLM, MARKUS, D. SCHMIDT und B. STEFFEN: *Model-Checking: A Tutorial Introduction.* In: A. CORTESI, G. FILE (Herausgeber): *Proc. of Static Analysis Symposium (SAS'99), Venice, Italy*, Band 1694 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 330–354, Heidelberg, Germany, September 1999. Springer-Verlag.

[60] MUCHNICK, STEVEN S.: *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers, San Francisco, California, 1997.

[61] NIELSEN, JAKOB: *Personalization is Over-Rated.* http://www.useit.com/alertbox/981004.html.

[62] NIELSON, F., H.R. NIELSON und C. HANKIN: *Principles of Program Analysis.* Springer Verlag, 1999.

[63] NIESE, OLIVER: *An Integrated Approach to Testing Complex Systems.* Dissertation, Universität Dortmund, Dortmund, Deutschland, 2003. http://eldorado.uni-dortmund.de:8080/FB4/ls5/forschung/2003/Niese.

[64] PAPOULIAS, ATHANASIOS, MARKUS STRAUCH und PETER LAMMICH: *EWIS Survey Fragebogen Howto.* Universität Dortmund, 2004.

[65] *Abschlussbericht PG 430 Interaktive Entwicklung personalisierter Webapplikationen*, April 2004. veranstaltet vom Lehrstuhl für Programmiersysteme und Übersetzerbau, Universität Dortmund.

[66] Reppel, Alexander: *Personalization and the Power to Control.* http://www.reppel.co.uk/relationship-marketing/personalization-and-the-power-to-control.html.

[67] Schiedermeier, Reinhard: *Programmieren mit Java – Eine methodische Einführung.* Pearson Studium, 2005.

[68] Schmidt, D. und B. Steffen: *Program Analysis as Model Checking of Abstract Interpretations.* In: *Proc. of Static Analysis Symposium (SAS'98), Pisa, Italy*, Band 1503 der Reihe *Lecture Notes in Computer Science (LNCS)*, Seiten 351–380, Heidelberg, Germany, September 1998. Springer-Verlag.

[69] Seidman, Robert: *Personally, On Personalization.* http://www.onlineinsider.com/html/august_24__1998.html.

[70] Stal, M.: *What characterizes a (software) component?* Band 19 der Reihe *Software, Concepts and Tools*, Seiten 49–56. 1998.

[71] Steffen, B.: *Data Flow Analysis as Model Checking.* In: *Proceedings of the International Conference on Theoretical Aspects of Computer Software, TACS'91*, Band 526 der Reihe *LNCS*, 1991.

[72] Steffen, B.: *Generating Data Flow Analysis Algorithms from Modal Specifications.* Science of Computer Programming, 21:115–139, 1993.

[73] Steffen, B. und T. Margaria: *The Agent Building Center: An Integrated Approach to Telephone and Internet Service Design.* In: *Electronics and Telecommunication Research Institute (ETRI), Taejon (S. Korea)*, 26. August 1997. half-day Tutorial.

[74] Steffen, B. und T. Margaria: *Coarse-grain Component Based Software Development: The* METAFrame *Approach.* In: *3. Fachkongress "Smalltalk und Java in Industrie und Ausbildung" (STJA'97), Erfurt (Germany)*, September 10-11 1997.

[75] Steffen, B. und T. Margaria: *METAFrame in Practice: Design of Intelligent Network Services.* In: E.-R. Olderog, B. Steffen (Herausgeber): *Correct System Design - Recent Insights and Advances*, Band 1710 der Reihe *Lecture Notes in Computer Science (LNCS) State-of-the-Art Survey*, Seiten 390–415. Springer-Verlag, Heidelberg, Germany, October 1999. Dedicated to Hans Langmaack on the occasion of his retirement from his professorship.

[76] Steffen, B. und T. Margaria: *METAFrame in Practice: Intelligent Network Service Design.* In: E.-R. Olderog, B. Steffen (Herausgeber): *Correct System Design – Issues, Methods and Perspectives*, Band 1710 der Reihe

Lecture Notes in Computer Science (LNCS), Seiten 390 – 415, Heidelberg, Germany, 1999. Springer-Verlag.

[77] Steffen, B., T. Margaria, V. Braun und N. Kalt: *Hierarchical Service Definition.* Annual Review of Communication, Int. Engineering Consortium (IEC), Chicago (USA), Seiten 847–856, 1997.

[78] Steffen, B., T. Margaria, A. Classen und V. Braun: *Incremental Formalization: A Key to Industrial Success.* SOFTWARE: Concepts and Tools, 17(2):78–91, July 1996. Springer-Verlag, Heidelberg, Germany.

[79] Steffen, B., T. Margaria, A. Classen, V. Braun und M. Reitenspiess: *An Environment for the Creation of Intelligent Network Services.* In: *"Intelligent Networks: IN/AIN Technologies, Operations, Services, and Applications - A Comprehensive Report"*, Seiten 287–300. International Engineering Consortium (IEC), Chicago, 1996.

[80] Steffen, B., T. Margaria, A. Classen, V. Braun, M. Reitenspiess und H. Wendler: *Service Creation: Formal Verification and Abstract Views.* In: *4th Int. Conf. on Intelligent Networks (ICIN'96), Bordeaux (France)*, Seiten 96–101, November 1996.

[81] Steffen, B. und E.-R. Olderog: *Formale Semantik und Programmverifikation.* In: P. Rechenberg, G. Pomberger (Herausgeber): *Informatik-Handbuch*, Kapitel A Theoretische Informatik, Seiten 129–148. Carl Hanser Verlag, 1997.

[82] Steffen, Bernhard: *METAFrame in Practice.* In: *ONR Workshop on Automated Formal Methods, Oxford (UK)*, June 19-21 1996.

[83] Steffen, Bernhard: *Telephone and Internet Services: A Fruitful Symbiosis.* In: *AIN'97, Chongi (S.Korea)*, August 27-29 1997. Invited talk.

[84] Stirling, C.: *Local Model Checking Games.* In: *Proc. 6th Intern.Conf. on Concurrency Theory (CONCUR'95)*, Band 962 der Reihe *Lecture Notes in Computer Science*, Seiten 1–11. Springer Verlag, 1995.

[85] Stirling, C. und P. Stevens: *Practical Model-Checking Using Games.* In: *TACAS 1998*, Band 1384 der Reihe *Lecture Notes in Computer Science*, Seiten 85–101, 1998.

[86] *Studieren im Netz - Virtuelles Studium und E-Learning an Hochschulen.* http://www.studieren-im-netz.de.

[87] SZYPERSKI, C.: *Component Sofware: Beyond object oriented programming.* Addison-Wesley, 1998.

[88] METAFrame *Agent Building Center Service Definition.* METAFrame Technologies GmbH, Dortmund.

[89] *Microsoft Office.* http://office.microsoft.com/.

[90] *OpenOffice.* http://www.openoffice.org/.

[91] TEMPLUS. http://templus.cs.uni-dortmund.de.

[92] TRETMANS, JAN und AXEL BELINFANTE: *Automatic Testing with Formal Methods.* In: *EuroSTAR'99: 7th European Int. Conference on Software Testing, Analysis and Review*, Barcelona, Spain, November 8–12 1999. EuroStar Conferences, Galway, Ireland.

[93] V. GRUHN, A. THIEL: *Komponentenmodelle.* Addison-Wesley, 2000.

[94] VARDI, MOSHE Y. und PIERRE WOLPER: *An Automata-Theoretic Approach to Automatic Program Verification.* In: *IEEE Symposium on Logic in Computer Science (LICS98)*, Seiten 332–344, June 1986.

[95] *Virtueller Campus Rheinland-Pfalz.* http://www.vcrp.de.

[96] *Virtuelle Hochschule Bayern.* http://www.vhb.org.

[97] *Virtuelle Universität Hagen.* http://www.fernuni-hagen.de/FeU/LrVU/.

[98] *Maßnahmen zur Verkürzung der Studienzeiten durch die Steigerung der Erfolgsquote im Informatik-Grundstudium: Antrag des Fachbereichs Informatik der Universität Dortmund Sofortprogramm zur Weiterentwicklung des Informatikstudiums an den deutschen Hochschulen (WIS)*, 2000.

[99] YOO, HAISEUNG: *N.N.* Dissertation, Universität Dortmund, Dortmund, Deutschland, noch nicht erschienen.

[100] *Zentrum für Fernstudium und universitäre Weiterbildung.* http://www.zfuw.de.