Universität Leipzig
Institut für Informatik
Abt. Betriebliche Informationssysteme

Bachelorarbeit

Integration des Elate-Aufgaben-Frameworks in die LMS-Plattform OLAT

Marvin Frommhold

marvin.frommhold@gmail.com

Studiengang B.Sc Informatik

Betreuer: Dr. Hans-Gert Gräbe, apl. Prof.

Leipzig, den 7. September 2009

Zusammenfassung

Diese Arbeit erläutert die Integration des Aufgaben-Frameworks in die LMS-Plattform OLAT in Form einer Erweiterung. Es wird ein Vorgehensmodell bestimmt, welches sich am besten für die Erstellung und zukünftige Entwicklung eignet. Anschließend werden die beiden Anwendungen und deren Erweiterungs- bzw. Integrationspunkte vorgestellt und analysiert. Es soll gezeigt werden, wie eine konkrete Integration verwirklicht werden kann und welche Besonderheiten dabei zu beachten sind. Außerdem werden Richtlinien und Prinzipien für das Versionsmanagement vorgegeben, welche im Zuge einer Weiterentwicklung nötig sind.

Inhaltsverzeichnis

1 Einleitung **5**

1.1 Zielsetzung . . . 5

1.2 Abgrenzung . . . 6

1.3 Gliederung . . . 6

2 Vorgehensmodell **8**

2.1 Agile Softwareentwicklung . . . 9

2.2 FDD - Feature-Driven-Development . . . 10

2.2.1 Prozessmodell . . . 10

2.2.2 Anwendung im Olate-Projekt . . . 11

3 Analyse **13**

3.1 Aufgaben-Framework . . . 13

3.1.1 Integration . . . 14

3.2 OLAT . . . 15

3.2.1 Erweiterungskonzept . . . 15

4 Entwurf **17**

4.1 Übersicht . . . 17

4.2 Komponenten . . . 17

4.2.1 OLAT-Erweiterung . . . 18

4.2.2 Datenmodell . . . 20

4.2.3 Persistierung . . . 21

4.2.4 Taskmodel . . . 23

4.3 Paketstruktur . . . 23

5 Implementierung **25**

5.1 ElateTest-Kursbaustein . . . 25

5.1.1 Zuordnung der Rollen . . . 26

5.2 Integration des Aufgaben-Frameworks . . . 27

5.2.1 Kommunikation 27
5.2.2 TaskFactory 28
5.2.3 Vorschaufunktion 32

6 Versionsmanagement **34**
6.1 Allgemeine Richtlinien 34
6.1.1 Konfiguration und Versionierung 35
6.1.2 Bereitstellung 36
6.2 Releasemanagement 36
6.2.1 Vorgehen 37
6.3 Fehlerverwaltung 37

7 Zusammenfassung **40**
7.1 Ausblick 41

Abbildungsverzeichnis **42**

Literaturverzeichnis **43**

1 Einleitung

Das Learning Management System OLAT wird seit dem Sommersemester 2007 am Lehrstuhl Betriebliche Informationssysteme eingesetzt. Mit dessen Hilfe werden die Lehrveranstaltungen der Abteilung organisiert und verwaltet. Im Laufe dieses Einsatzes haben sich einige neue Anforderungen herausgebildet, welche nicht durch das Grundsystem abgedeckt werden. Infolge dessen ist ein Entwicklerkreis entstanden, welcher diese Bedürfnisse in Form von Erweiterungen umsetzt.

Das bedeutendste Einsatzgebiet betrifft hier vor allem die Unterstützung der operativen Prozesse bei der Verwaltung und Ausführung von Prüfungen. So ist im Rahmen des XMAN-Projektes eine Prüfungsverwaltung entstanden, welche seit dem Sommersemester 2008 erfolgreich am Lehrstuhl eingesetzt wird [5].

Aktuell besteht eine weiterer Wunsch, welcher eine Anbindung des elatePortal-Projekts vorsieht. Im Speziellen soll es möglich sein, Taskmodel-Tests direkt in OLAT zu verwalten und durchzuführen. Das Projekt, welches eine Realisierung dessen vorsieht, wird im Folgenden mit *Olate* bezeichnet.

1.1 Zielsetzung

Ziel dieser Arbeit ist eine Integration des am Institut für Informatik der Universität Leipzig entwickelten Aufgaben-Frameworks, als Teil des elatePortal-Projekts, in das Learning-Management-System OLAT. Es wird die Entwicklung einer Erweiterung beschrieben, mit deren Hilfe sich Taskmodel-Tests in OLAT ausführen und verwalten lassen. Bei der Erstellung geht es im Besonderen um die Berücksichtigung der zukünftigen Entwicklung, um eine dafür geeignete Projektverwaltung und die Einbindung in die Konfigurationsdatenbank der Abteilung Betriebliche Informationssysteme. Dabei ergeben sich folgende Fragestellungen, welche durch diese Arbeit beantwortet werden sollen.

(1) Welches Entwicklungsmodell eignet sich für eine solche Erweiterung?

(2) Wie kann eine konkrete Integration realisiert werden?

(3) Wie soll die zukünftige Entwicklung ablaufen? Was passiert bei Versionsänderungen der verwendeten Anwendungen?

1.2 Abgrenzung

Die zu erstellende Erweiterung richtet sich in erster Linie nach den Funktionalitäten des ebenfalls im elatePortal-Projekts entstandenen ExamServer. Dieser dient zur Authentifikation und Autorisierung der Teilnehmer und stellt eine Verwaltung für Taskmodel-Tests zur Verfügung. Diese Funktionalitäten sollen am Ende den Nutzern der Olate-Erweiterung zur Verfügung stehen. Es ist nicht vorgesehen, zusätzliche Funktionen bereitzustellen. Es wird jedoch speziell beim Entwurf darauf geachtet, dass das Projekt zu einem späteren Zeitpunkt einfach geändert bzw. erweitert werden kann.

1.3 Gliederung

Bei der Olate-Erweiterung handelt es sich um eine Zusammenführung zweier verschiedener Projekte, woraus sich besondere Ansprüche an ein zu wählendes Entwicklungsmodell ergeben. Im Abschnitt 2 werden diese Anforderungen konkretisiert und geeignete Entwicklungsmodelle vorgestellt. Aus dieser Betrachtung heraus wird ein probates Modell ausgewählt und die spezielle Anwendung dessen im Entwicklungsprozess erörtert.

Im Abschnitt 3 werden die beiden Projekte, welche als Ziel dieser Arbeit miteinander verbunden werden, genauer vorgestellt. Dabei wird vor allem auf die Erweiterungs- und Integrationspunkte eingegangen. Es wird gezeigt, welche Schnittstellen das Aufgaben-Framework anbietet und wie diese genutzt werden können, um es in eine Fremdanwendung zu integrieren. Für die LMS-Plattform OLAT wird das Erweiterungskonzept vorgestellt und welche Anforderungen sich daraus für das Olate-Projekt ergeben.

Abschnitt 4 befasst sich mit dem Entwurf des Olate-Projekts. Es werden Details zu den zu erstellenden Funktionalitäten und zum Aufbau gegeben. Des Weiteren

werden die verwendeten Prinzipien und Muster vorgestellt, sowie Designentscheidungen erläutert.

Die Implementierung und deren Besonderheiten werden im Abschnitt 5 behandelt. Ein deutliches Augenmerk wird dabei auf die Realisierung der Integrationsschnittstelle des Aufgaben-Frameworks gelegt, da für diese bisher nur spezifische Implementierungen vorhanden sind, aber keine allgemeine Dokumentation vorliegt.

Für den zukünftigen Einsatz der Erweiterung ist es sehr wahrscheinlich, dass sich Anforderungen und/oder die beiden verwendeten Projekte ändern. Folglich muss auch die Olate-Erweiterung angepasst bzw. um neue Funktionen erweitert werden. Wie diese Änderungen geplant und umgesetzt werden sollen, wird im Abschnitt 6 dargelegt.

Im letzten Abschnitt werden die Ergebnisse dieser Arbeit noch einmal kurz zusammengefasst. Außerdem wird ein Ausblick über etwaige zukünftige Entwicklungen im Olate-Projekt gegeben.

2 Vorgehensmodell

Um die Komplexität eines Softwareproduktes und dessen Entwicklung beherrschbar zu machen, verwendet man einen Plan, in der Softwareentwicklung Vorgehensmodell genannt. Vorgehensmodelle teilen den Entwicklungsprozess in einzelne Phasen auf und legen für jede dieser Phasen durchzuführende Aktivitäten fest. Im folgenden wird anhand der Gegebenheiten des Olate-Projektes ein geeignetes Vorgehensmodell diskutiert und vorgestellt. Dieses soll insbesondere auch die Phasen nach der Fertigstellung des Projekts im Rahmen dieser Arbeit mit einbeziehen.

Bei der Olate-Erweiterung handelt es sich um sehr kleines Projekt mit wenigen Entwicklern. Es werden zwei Systeme miteinander integriert, welche sich fortwährend in der Entwicklung befinden und dadurch regelmäßig API[1]- und Funktionalitätsänderungen stattfinden. Durch diese ständigen Änderungen ergeben sich auch immer wieder neue Anforderungen an das Projekt, welche im Voraus nicht festgelegt werden können, sondern sich im Laufe der Entwicklung herausbilden. Diese Gegebenheiten sollten in besonderer Weise von dem verwendeten Vorgehensmodell unterstützt werden. Folgende Anforderungen sollte das zu verwendente Modell erfüllen:

- regelmäßige Reviews zwischen Entwicklern und dem Kunden
- flexible Planung und Berücksichtigung von sich ändernden Anforderungen
- knappe Phasen und daraus resultierende kurze Zyklen
- Planung eines zeitlich unbestimmten Entwicklungsprozesses
- geringer Verwaltungsaufwand, Konzentration auf das Entwickeln

Die klassischen Modelle, wie zum Beispiel das Wasserfall- oder das Spiralmodell, eignen sich nicht für solch ein Projekt. Zum einen legen diese einen relativ starren Entwicklungsplan fest, bei dem vor allem die Anforderungen schon vor der

[1] Application Programming Interface

eigentlichen Programmierarbeit feststehen müssen, zum anderen würde die Projektplanung und -verwaltung, im Speziellen für die Erstellung der verschiedenen Artefakte aus den einzelnen Phasen, den Aufwand auf Grund der geringen Ressourcen unrealistisch in die Höhe treiben. Somit fallen die klassischen Modelle aus dieser Betrachtung heraus.

2.1 Agile Softwareentwicklung

Aus den zuvor genannten Problemen hat sich zu Anfang der 1990er Jahre ein neuer Ansatz entwickelt - die Agile Softwareentwicklung. Das Ziel besteht darin, den Softwareentwicklungsprozess schlanker und flexibler zu gestalten. Ein erster Vertreter und das wohl bekannteste agile Vorgehensmodell für die Anwendung agiler Prozesse stellt Extreme Programming, kurz XP, dar.

> *„XP ist eine leichte, effiziente, risikoarme, flexible, kalkulierbare, exakte und vergnügliche Art und Weise der Softwareentwicklung.“*

So beschreibt es Kent Beck in seinem Buch *Extreme Programming - Das Manifest* [3]. XP unterscheidet sich gegenüber den klassischen Methoden durch kurze Zyklen mit konkreten, fortwährenden Feedbacks. Zudem bietet es einen inkrementellen Planungsansatz, womit ein allgemeiner Plan, der auch eine zukünftige Entwicklung berücksicht, erstellt wird. XP berücksichtigt auch im besonderen Maße sich ändernde Anforderungen und wendet diesbezüglich einen evolutionären Designprozess an. Jedoch gestaltet sich XP als zu starr, was die durchzuführenden Prinzipien und Praktiken betrifft. So wird vorgeschrieben, dass das Programmieren in Paaren erfolgen soll, was im Zuge dieser Arbeit nicht möglich ist. Außerdem müssen fortwährend automatisierte Tests für alle Komponenten und Einheiten geschrieben und angewandt werden. Im Bezug auf die knappen Ressourcen und Möglichkeiten, welche die beiden zu integrierenden Anwendungen bieten, ist dies aber nicht immer und an allen Stellen möglich. Das folgende Zitat aus dem Buch soll daher belegen, warum ein Einsatz von Extreme Programming für dieses Projekt nicht in Frage kommt:

> *„XP ist eine Disziplin [...], weil es bestimmte Dinge gibt, die man tun muss, wenn man XP einsetzen will.“*

Im Folgenden soll nun ein weiteres agiles Vorgehensmodell betrachtet werden, welches sich durch seine Kompaktheit und Ausrichtung als geeigneter Kandidat für eine Anwendung im Rahmen des Olate-Projektes hervorhebt.

2.2 FDD - Feature-Driven-Development

Feature-Driven-Development, zu deutsch Funktionsgesteuerte Entwicklung, wurde 1997 von Jeff De Luca definiert [8]. Es ist eine sehr schlanke Methode und stellt den Feature-Begriff in den Mittelpunkt. Jedes Feature stellt einen Mehrwert für den Kunden dar. Aus diesem Grund wird die Entwicklung anhand eines Feature-Plans organisiert. Des Weiteren werden Rollen und deren Aufgaben definiert.

2.2.1 Prozessmodell

Klassische FDD-Projekte durchlaufen fünf Prozesse, wobei die ersten drei einmal ausgeführt und die letzten beiden für jedes einzelne Feature wiederholt werden.

Prozess 1: Entwickeln eines Gesamtmodells
Es wird der Inhalt und Umfang des zu entwickelnden Systems mit allen Projektbeteiligten definiert und erste Systembereiche festgelegt. Ziel ist ein gemeinsamer Konsens über das gesamte System, sowie das fachliche Kernmodell.

Prozess 2: Erstellen einer Feature-Liste
Nun werden die im ersten Prozess festgelegten Systembereiche durch die Chefprogrammierer verfeinert und die einzelnen Funktionen in einer Feature-Liste festgehalten. Ein Feature stellt dabei eine granulare Funktion dar und sollte nach Möglichkeit mit folgendem Schema notiert werden:

<Aktion><Ergebnis><Objekt>

Zum Beispiel wäre das „Berechnen der Gesamtsumme aller Artikel“ eine korrekte Notation eines Features nach diesem Schema.

Prozess 3: Erstellen eines Feature-Plans

Anschließend werden die Features durch den Projektleiter, Entwicklungsleiter und die Chefprogrammierer in eine Reihenfolge gebracht, in der sie realisiert werden sollen. Dabei müssen die Abhängigkeiten zwischen den Features aufgelöst, die Auslastung der Programmierteams sowie die Komplexität der einzelnen Features berücksichtigt werden.

Prozess 4: Entwurf je Feature

Die Entwicklerteams erstellen für ihre jeweiligen zu realisierenden Features einen Entwurf. Es werden außerdem erste Klassen- und Methodenrümpfe geschrieben. Anschließend werden die Ergebnisse inspiziert und eventuell korrigiert.

Prozess 5: Implementierung je Feature

Jetzt werden die vorbereiteten Features implementiert. In diesem Zusammenhang werden Komponententests und Code-Inspektionen zur Qualitätssicherung eingesetzt.

2.2.2 Anwendung im Olate-Projekt

Auf Grund der Entwicklung in einzelnen Features werden kurze Phasen erreicht. Die Entwickler und der Kunde können dadurch in kleinen Schritten nachverfolgen, in welche Richtung das Projekt verläuft. Es können notwendige Anpassungen frühzeitig erkannt und eingearbeitet werden. Durch die einfache Einteilung in Features kann zudem ein allgemeiner Projektplan aufgestellt werden, der für einen zeitlich unbestimmten Entwicklungsprozess geeignet ist. Dies ist insbesondere für das Versionsmanagement von Bedeutung. Ein weiterer Vorteil solch einer feinen Granularität ist die leichte Aufteilung der durchzuführenden Tätigkeiten auf verschiedene Entwickler. Im Falle dieser Arbeit wurden alle Features durch einen Entwickler realisiert. Ändert sich die Anzahl der Projektmitglieder, wird entsprechend aufgeteilt.

FDD erfüllt somit im ausreichenden Maße alle Anforderungen und soll im Olate-Projekt Anwendung finden. Dennoch muss das Vorgehen an einige Besonderheiten angepasst werden. So findet zum Beispiel keine Rückkopplung zu vorherigen Prozessen statt. Dies ist aber von essentieller Bedeutung. Werden etwaige Desginfehler festgestellt, müssen die entstandenen Artefakte aus den früheren Prozessen

geändert bzw. angepasst werden. Aus diesen Überlegungen ergibt sich das in Abbildung 2.1 gezeigte Prozessmodell.

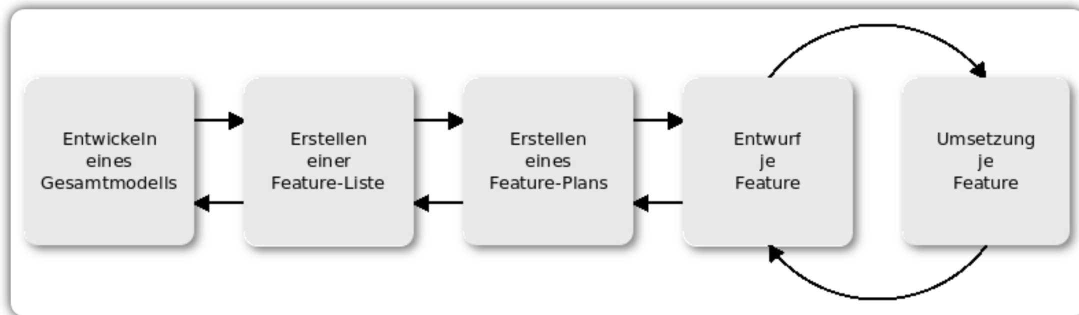

Abb. 2.1: Diese Grafik zeigt das für das Olate-Projekt angepasste FDD-Prozessmodell.

Die ersten drei Prozesse werden nach dem klassischen FDD-Vorgehen durchgeführt. So wird ein Gesamtmodell und eine Feature-Liste erstellt, siehe Abschnitt 4. Anschließend wird ein Ablaufplan erstellt und die Features einzeln modelliert und implementiert, siehe Abschnitt 5. Das zukünftige Vorgehen, zum Beispiel Anpassungen bei neuen Versionen der zugrundeliegenden Systeme, wird im Abschnitt 6 erläutert.

3 Analyse

Im Folgenden soll zuerst analysiert werden, wie sich OLAT und das Aufgaben-Framework geeignet zusammen einsetzen lassen. Im Anschluss daran werden die beiden zu integrierenden Anwendungen selbst und deren Möglichkeiten zur Erweiterung und Integration vorgestellt.

3.1 Aufgaben-Framework

Ein zentrales Artefakt im elatePortal-Projekt[1] stellt das Aufgaben-Framework, auch Taskmodel genannt, dar [4]. Es bietet eine umfassende Unterstützung für die Durchführung von elektronischen Prüfungen. So stellt es Funktionen zum Erstellen von Fragenpools, randomisierten Zusammenstellungen von Prüfungen und besonders die semiautomatische Korrektur von den Aufgaben bereit. Das heißt, nicht automatisch auswertbare Aufgaben werden Korrektoren zugeordnet und von diesen wiederum mit Hilfe des Frameworks kontrolliert.

Aufgebaut ist das System aus drei Teilen. Einer API (Taskmodel-API), welche die Struktur beschreibt und den Zugriff auf die Implementierungen kapselt. Darauf aufbauend eine Referenzimplementierung (Taskmodel-CORE) unter Verwendung des Spring-Frameworks[2], in der alle Interfaces implementiert werden und eine Unterstützung für die benötigten Aufgabentypen geboten wird. Der dritte Teil ist die eigentliche Webapplikation (Taskmodel-CORE-VIEW), welche die Interaktion mit dem Nutzer durchführt.

Das Aufgaben-Framework steuert selbstständig den gesamten Lebenszyklus einer in Ausführung befindlichen Aufgabe, auch *Tasklet* genannt. Letzteres ist die Zusammenführung eines Kandidaten und einer Aufgabendefinition, welche wiederum als *TaskDef* bezeichnet wird. Diese Definition stellt den eigentlichen Test dar, das

[1] `http://www.elateportal.de`
[2] `http://www.springsource.org/`

heißt die Fragen und wie diese bewertet werden. In den einzelnen Phasen des Lebenszyklus eines Tasklets (siehe Abb. 3.1) stehen unterschiedliche Funktionen zur Verfügung. So kann eine Aufgabe erst kontrolliert werden, wenn sie sich im Zustand *processed* befindet. Während einer Kontrolle können zusätzlich Kommentare angegeben werden. Ist eine Aufgabe fertig kontrolliert, also im Zustand *corrected*, hat der Testkandidat nochmals die Möglichkeit, die korrigierte Prüfung einzusehen und Bemerkungen zu hinterlassen, welche wiederum vom Korrektor bestätigt werden müssen.

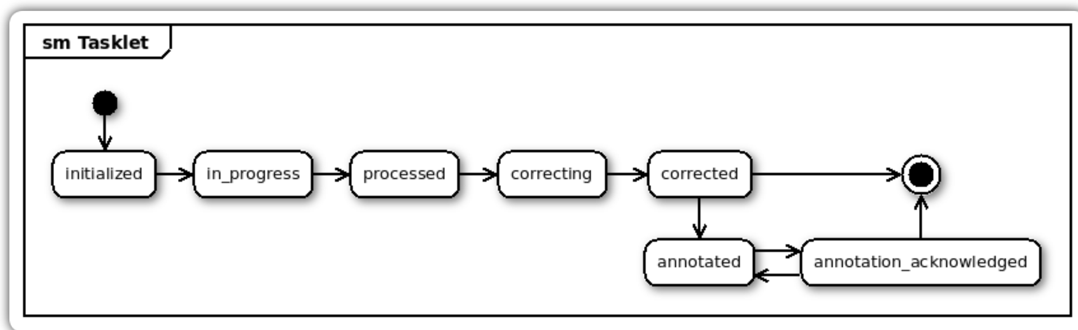

Abb. 3.1: Dieses Zustandsdiagramm zeigt den Lebenszyklus eines Tasklets.

3.1.1 Integration

Bei der Entwicklung des Aufgaben-Frameworks wurde die Möglichkeit der Integration in andere Systeme bereits bedacht. So sind die Persistenzaufgaben in einer Komponente `TaskFactory` gekapselt, welche vom jeweiligen Host-System implementiert werden muss. Diese Schnittstelle ist ein wichtiger Bestandteil der API und wird von der Webapplikation beispielsweise zum Erstellen, Laden und Speichern von Tasklets bzw. TaskDefs genutzt. In den Abschnitten 4.2.4 und 5.2.2 wird noch einmal genauer auf diese Komponente und deren Umsetzung in diesem Projekt eingegangen.

Weiterhin besitzt die Referenz-Implementierung des Aufgaben-Frameworks bereits einen Mechanismus zur Persistierung von Tasklets und TaskDefs. Diese Implementierung wurde mit Hilfe von JAXB[3], einer Programmschnittstelle in Java, um Daten aus einer XML-Schema-Instanz automatisch an Java-Klassen zu binden, realisiert.

[3]Java Architecture for XML Binding - `https://jaxb.dev.java.net/`

3.2 OLAT

OLAT[4] steht für „Online Learning and Training“ und ist ein Web-basiertes, Open-Source Learning Management System (LMS). Es basiert auf der Programmiersprache Java und wird seit 1999 an der Universität Zürich entwickelt und eingesetzt. Es bietet eine umfassende Unterstützung zur Verwaltung und Bereitstellung von Lerninhalten. Neben einem flexiblen Kurssystem, einer Unterstützung von gängigen E-Learning-Standards, der Bereitstellung von Groupwaretools, wie Kalender, Foren, Wikis etc., und einer umfassenden Benutzerverwaltung, wurde im Hinblick auf ein breites Anforderungsspektrum besonderer Wert auf eine hochgradig modulare, leicht anpassbare und erweiterbare Architektur gelegt. Weiterführende Informationen können der OLAT 6 Funktionsübersicht [2] entnommen werden.

3.2.1 Erweiterungskonzept

OLAT besitzt ein umfangreiches Erweiterungskonzept. Das Hauptziel ist dabei die Möglichkeit, den Funktionsumfang an die eigenen Bedürfnisse anzupassen bzw. auszubauen, ohne den originalen Quellcode verändern zu müssen. Erweiterungen werden in Form eines Java-Archivs[5] in den Ordner `WEB-INF/lib` gelegt und über die Konfigurationsdatei `olat_extensions.xml` eingebunden. Jede Erweiterung muss die Schnittstelle `org.olat.core.extensions.Extension` implementieren, über die das System die neuen Komponenten für die jeweiligen zu erweiternden Stellen lädt.

Das Erweiterungskonzept ist aus mehreren Erweiterungspunkten zusammengesetzt. Es besitzt aber nicht jede Komponente einen eigenen Erweiterungspunkt. Es folgt eine kurze Übersicht der erweiterungsfähigen Stellen:

`org.olat.persistence.DB`
: Über diesen Punkt kann die Datenbank um neue zu persistierende Objekte erweitert werden.

`org.olat.home.HomeMainController`
: Es können weitere Menüpunkte zur Home-Seite hinzugefügt werden.

[4] `http://www.olat.org`
[5] eine ZIP-Datei mit der Dateiendung `.jar` zur Verteilung von Java-Klassenbibliotheken und -Programmen

`org.olat.dispatcher.DispatcherAction`
: Durch einen neuen Mapper kann ein komplett neuer Arbeitsablauf für einen bestimmten URL-Pfad hinzugefügt werden.

`org.olat.gui.components.Window`
: Erweiterung um neue, eigene Cascading Style Sheets (CSS) zur Anpassung der Darstellung.

`org.olat.gui.control.generic.dtabs.DTabs`
: Hinzufügen neuer Reiter zur Hauptoberfläche.

Das System der Kursbausteine ist ein weitere Stelle, welche auf Grund seiner Umsetzung ebenfalls ohne Änderung der Originalquellen erweiterbar ist, jedoch keine Erwähnung in der Entwickler-Dokumentation findet. Diese Möglichkeit der Erweiterung wird zur Einbindung der Tests in Kursen verwendet und stellt die zentrale Komponente des Olate-Projektes dar. Eine genauere Erläuterung zur Umsetzung solch eines Kursbausteins gibt der Abschnitt 4.2.1.

4 Entwurf

4.1 Übersicht

Da von Anfang an das Ziel bestand, die OLAT-Quellen für die Realisierung nicht zu verändern, erfolgt die Integration des Aufgaben-Frameworks durch das Hinzufügen eines neuen Kursbausteins zum bisherigen Kurssystem von OLAT, unter der Verwendung des Erweiterungsmechanismus, wie in 3.2.1 beschrieben. Eine Instanz eines solchen Bausteins in einem Kurs repräsentiert genau einen Test[1]. Somit lassen sich mehrere verschiedene Tests (Instanzen) in einen Kurs integrieren. Das Einstellen erfolgt durch die üblichen Schritte. Im Kurseditor wird ein neuer Baustein hinzugefügt und der Test, repräsentiert durch eine XML-Datei, wird in das System hochgeladen. Zusätzlich können noch weitere Angaben wie ein Titel, eine kurze Beschreibung usw. gemacht werden.

Im Folgenden wird zwischen zwei Rollen unterschieden. Als Testkandidaten werden diejenigen Nutzer einer OLAT-Instanz bezeichnet, welche das Recht haben, einen Test abzulegen. Korrektoren sind wiederum jene Nutzer, welche das Recht haben, einen abgelegten Test[2] zu korrigieren. Der Abschnitt 5.1 beschreibt, wie eine Zuordnung von OLAT-Nutzern zu diesen Rollen erfolgt.

4.2 Komponenten

Die Erweiterung besteht aus vier Komponenten bzw. Systembereichen. Jede Komponente bildet dabei einen Teil der Funktionen der gesamten Erweiterung ab. Eine nähere Erläuterung der einzelnen Komponenten folgt in den kommenden Abschnitten. Es werden auch die für jeden Bereich zu erfüllenden Features aufgezählt.

[1] bezeichnet eine komplette Aufgabenstellung, d.h. ein TaskDef
[2] Status *processed*

4.2.1 OLAT-Erweiterung

Diese Komponente richtet sich streng nach dem Erweiterungskonzept von OLAT (siehe Abschnitt 3.2.1).

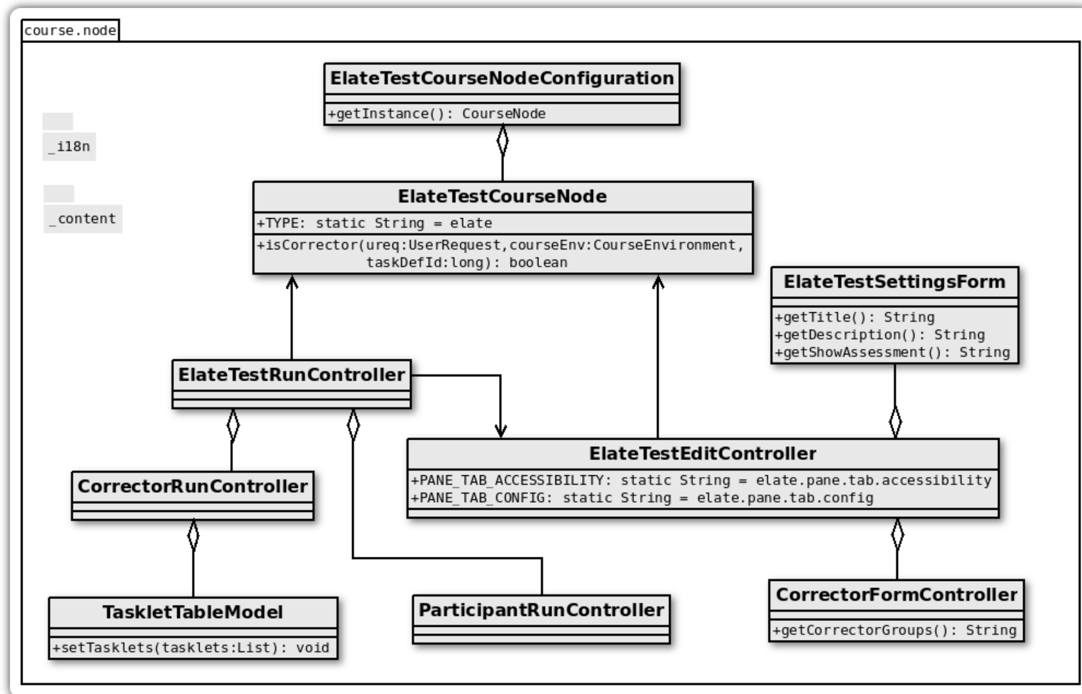

Abb. 4.1: Aufbau des Kursbausteins „ElateTest“

Um einen neuen Kursbaustein bereitzustellen, müssen mehrere Schnittstellen implementiert werden. Zum einen sind das die beiden Interfaces `CourseNodeConfiguration` und `OLATExtension`, welche zusammen in der Klasse `ElateTestCourseNodeConfiguration` umgesetzt sind. Diese gibt an, wie der Kursbaustein zu integrieren ist, stellt benötigte Ressourcen, wie zum Beispiel CSS-Definitionen[3] zur Verfügung, und ist für die Erzeugung von Instanzen des Kursbausteins verantwortlich. Diese Instanzen werden durch die Klasse `ElateTestCourseNode` repräsentiert, welche die abstrakte Klasse `AbstractAccessableCourseNode` ergänzt und erweitert. Durch die Verwendung dieser sind bereits einige Standardfunktionen umgesetzt und müssen nicht selbst implementiert werden. Eine `CourseNode` stellt die nötigen Steuerungsklassen (`Controller`) zum Editieren und Ausführen bereit. Außerdem wird hier die Konfiguration des Bausteins validiert, Import-, Export- und Kopierfunktionen angeboten und die Zugriffskontrolle durchgeführt. Der `ElateTestRunController`, welcher für die Ausführung bzw. Anzeige eines

[3]Cascading Style Sheets: deklarative Sprache zur Strukturierung von Dokumenten

Elate-Kursbausteins verantwortlich ist, verfügt über zwei verschiedene Sichten. Eine Sicht für die Kandidaten eines Tests, wiederum repräsentiert durch den `ParticipantRunController`, und eine Sicht für die Korrektoren, welche durch den `CorrectorRunController` abgebildet wird. Die benötigten Funktionen für den Kurseditor werden durch den `ElateTestEditController` bereitgestellt. Dieser bietet die Möglichkeit ein TaskDef in Form einer XML-Serialisierung hochzuladen. Die `ElateTestSettingsForm` bietet die Möglichkeit, weitere Angaben, wie zum Beispiel einen Titel und eine Beschreibung, machen zu können. Der `CorrectorFormController` dient zur Angabe der Gruppen, deren Betreuer die Korrektoren des Tests darstellen.

Um die unter 3.2.1 genannten Punkte erweitern zu können, muss zuerst eine allgemeine Erweiterung erstellt werden, welche das Interface `Extension` implementiert. Diese, repräsentiert durch die Klasse `OlateExtension`, stellt dann die spezifischen Ergänzungen für die einzelnen Punkte bereit. Damit die Datenmodell-Komponente Objekte in der OLAT-Datenbank speichern kann, muss die Persistenzschicht ergänzt werden. Dies geschieht durch Hibernate-Mapping-Dateien[4]. OLAT durchsucht beim initialen Starten selbstständig den Klassenpfad nach solchen Mappings und bindet diese ein. Da aber die Olate-Erweiterung als Jar-Datei ausgeliefert wird und somit die darin enthaltenen Mappings sich nicht im `WEB-INF/classes`-Ordner befinden, werden diese nicht automatisch mit eingebunden. Aus diesem Grund müssen die Mappings manuell hinzugefügt werden. Dies wird durch die Klasse `OlateHibernateConfigurator` realisiert.

Features

Folgende Features ergeben sich für das Erweitern von OLAT um einen neuen Kursbaustein „ElateTest“ und das Einbinden neuer Hibernate-Mappings:

- Erstellen eines neuen Kursbausteins
- Implementieren der Editierfunktion des Kursbausteins
- Implementieren einer Sicht für Testkandidaten
- Implementieren einer Sicht für Korrektoren
- Unterstützung der Vorschaufunktion eines Kurses

[4] `https://www.hibernate.org/`

- Erstellen einer neuen Extension
- Erweiterung der Persistenzschicht

4.2.2 Datenmodell

Um die agile Softwareentwicklung zu unterstützen, wird die Implementierung gegen Interfaces durchgeführt. Dadurch wird eine leichte, aber kontrollierte Kopplung erreicht, was ein Prinzip der Kapselung darstellt. Damit kann im Entwicklungsprozess und darüber hinaus auf Fehler oder Veränderungen ohne großen Aufwand reagiert und die betroffene Klasse im Notfall einfach ausgetauscht werden. Das gesamte Modell ist dabei an die Objekte aus dem Taskmodel angelehnt.

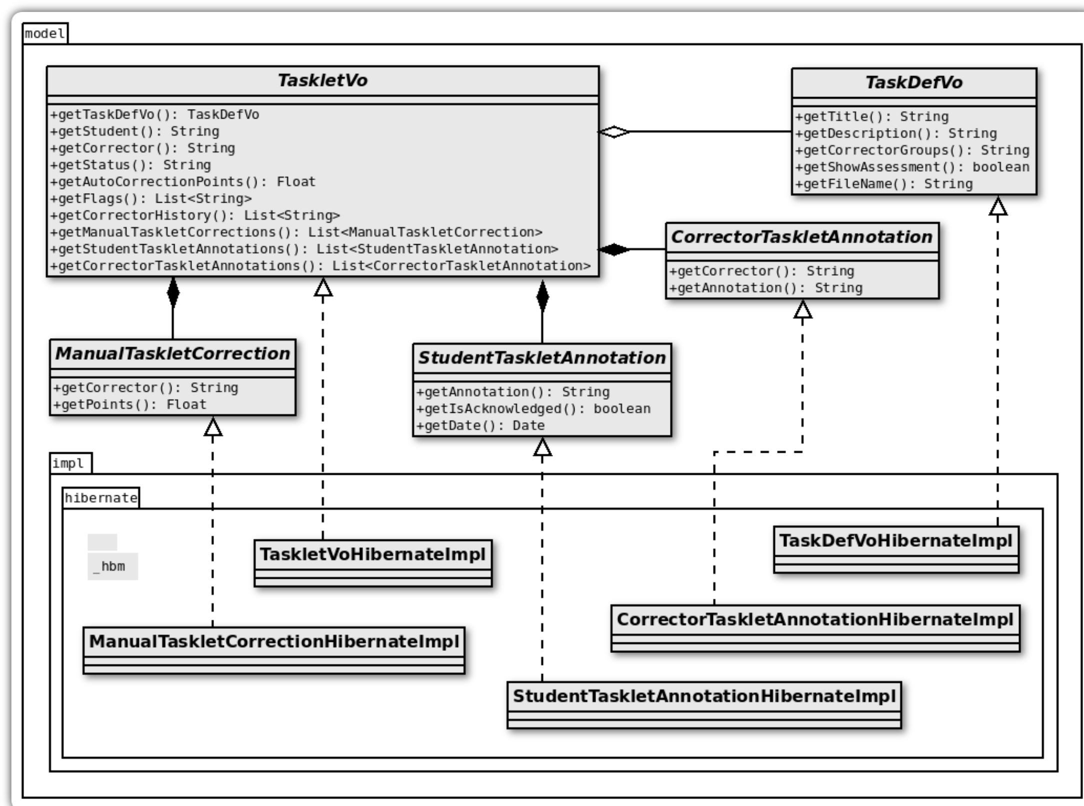

Abb. 4.2: Aufbau und Klassen des Datenmodells.

Ein Test und dessen zugeordneten Informationen (Meta-Daten) werden durch das Interface `TaskDefVo`, auch TaskDef-Datenobjekt genannt, abgebildet. Wie bereits im Abschnitt 3.1 erläutert, stellt ein Tasklet einen Test bzw. eine Aufgabenstellung in Bearbeitung dar. Jedem Test in der Bearbeitung ist ein Testkandidat zugeordnet. Des Weiteren ist jedem Tasklet genau ein Test (TaskDef) zugewiesen.

Diese und andere Informationen, wie der Status eines Tasklets und die erreichte Punktzahl werden durch das Interface `TaskletVo`, auch Tasklet-Datenobjekt, modelliert. Außerdem können Korrektoren während der Korrektur Kommentare abgeben. Testkandidaten wiederum haben die Möglichkeit einen fertig korrigierten Test zu kommentieren. Diese Anmerkungen werden zusammen mit der Identität des Erstellers durch die beiden Interfaces `CorrectorTaskletAnnotation` und `StudentTaskletAnnotation` repräsentiert und in Listen im `TaskletVo` gespeichert. Eine weitere Liste enthält alle manuell durchgeführten Korrekturen, abgebildet durch das Interface `ManualTaskletCorrection`.

Die in dieser Erweiterung verwendeten Datenobjekte werden mit Hilfe des OLAT-eigenen Persistenzmechanismus abgespeichert. Dafür beinhaltet diese Komponente eine Referenz-Implementierung, inklusive Mapping-Dateien.

Features

Laut angewandtem Entwicklungsmodell ergeben sich für das Datenmodell folgende Features:

- Erstellen der Interfaces für die Datenobjekte
- Erstellen einer Referenzimplementierung für die Verwendung mit Hibernate
- Erstellen der Hibernate-Mapping-Dateien

4.2.3 Persistierung

Bei der Persistierung der Daten findet eine Mischung aus verschiedenen Strategien statt. So werden die Meta-Daten[5] und die Tasklets bzw. TaskDefs unterschiedlich abgespeichert. Jedoch wird, um einen einheitlichen Stil und eine bessere Austauschbarkeit zu erreichen, das *Data Access Object Pattern* [9] angewandt. Zusätzlich wenden diese, kurz DAOs genannt, das CRUD-Prinzip an. CRUD ist ein Akronym und umschreibt die grundlegenden Datenoperationen `Create` (Datensatz erstellen), `Retrieve` (Datensatz lesen), `Update` (Datensatz aktualisieren) und `Delete` (Datensatz löschen). Des Weiteren werden die DAOs über eine zentrale Stelle, die `DaoFactory`, erzeugt. Somit kann die komplette Implementierung

[5] Daten, welche durch das Datenmodell repräsentiert und getrennt vom Tasklet bzw. TaskDef gespeichert werden

ausgetauscht werden, ohne die restlichen Komponenten ändern zu müssen. Eine spätere Änderung der Persistenzstrategie ist damit ohne Probleme möglich.

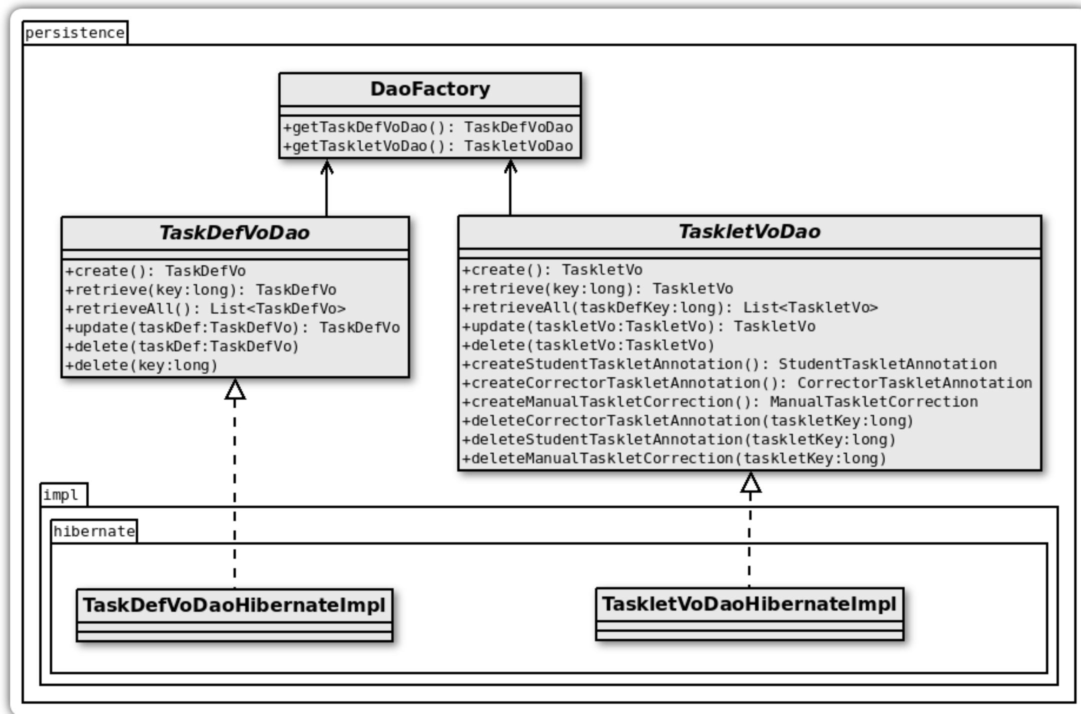

Abb. 4.3: Die Persistierung erfolgt über Data Access Objects, welche über eine Factory erzeugt werden.

Die Speicherung der Meta-Daten aus dem Datenmodell erfolgt über die jeweiligen Data Access Objects (Abb. 4.3). Da OLAT bereits einen eigenen Persistenzmechanismus besitzt und dieser sich auch erweitern lässt, wird dieser verwendet. Die Komponente stellt deswegen eine Referenzimplementierung, welche die OLAT-Datenbankschicht nutzt, bereit.

Die Tasklets und TaskDefs werden jedoch nicht in der Datenbank abgelegt. Da diese beiden Objekte sehr viele Informationen beinhalten können und das Aufgaben-Framework bereits eine Realisierung zur Speicherung dieser Objekte in XML-Dateien (Abschnitt 3.1) enthält, werden diese nicht in der OLAT-Datenbank abgelegt. Stattdessen werden sie im Datenverzeichnis von OLAT im jeweiligen Kursordner in einem neuen Unterordner `elatenodes` gespeichert.

Features

Für die Erstellung der Persistenz-Komponente sind folgende Features zu implementieren:

- Erstellen der `DaoFactory`
- Erstellen der Interfaces für die Data Access Objects
- Implementieren der DAOs unter Verwendung der OLAT-Persistenzschicht

4.2.4 Taskmodel

Zur Anzeige und Durchführung der Tests ist das Aufgaben-Framework verantwortlich. Um mit dieser eigenständigen Webapplikation kommunizieren zu können, stellt sie eine Schnittstelle bereit. Diese muss vom Host-System umgesetzt werden. Dies geschieht durch die Klasse `TaskFactoryOlatImpl`, welche das Interface (`de.-thorstenberger.taskmodel.TaskFactory`) implementiert. Über dieses Fabrik-Objekt erhält das Aufgaben-Framework alle nötigen Daten. Um eine Vorschaufunktion zu ermöglichen, muss zusätzlich zur `TaskFactory` die Referenzimplementierung des `TaskletContainer` durch eine eigene, `TaskletContainerOlatImpl`, ausgetauscht werden. Weitere Informationen hierzu und eine genaue Beschreibung zur Umsetzung der Anbindung folgt im Abschnitt 5.2.

Features

Es ergeben sich für diese Komponente zwei zu realisierende Features:

- Implementieren der `TaskFactory`
- Implementieren des `TaskletContainers`

4.3 Paketstruktur

Die Paketstruktur ist an die genannten Komponenten angelehnt. Es gibt vier Pakete, welche jeweils ein Komponente repräsentieren.

Die Klassen, welche die OLAT-Erweiterungspunkte implementieren, befinden sich im Paket `de.unileipzig.olate.olat`. Dabei sind alle Bezeichnungen der einzelnen Klassen und Unterordner an das Namensschema von OLAT angelehnt (`*Controller`, `*Form`, ...). Im Unterordner `course.node` sind sämtliche Klassen enthalten, welche zur Bereitstellung eines neuen Kursbausteins benötigt werden. Die Ressourcen, wie Sprachdateien für die Übersetzung und Velocity-Vorlagen, sind in den OLAT-typischen Verzeichnissen `_i18n` und `_content` untergebracht. Der Unterordner `extensions` beinhaltet wiederum das Extension-Modul und die Klasse zur Erweiterung der Persistenzschicht.

Das Datenmodell (`de.unileipzig.olate.model`) und die Persistierung (`de.unileipzig.olate.persistence`) sind ebenfalls getrennt untergebracht. In beiden Paketen gibt es aber eine einheitliche Struktur. Sämtliche Interfaces sind direkt in den Ordnern abgelegt. In einem Unterverzeichnis `impl` befinden sich wiederum die spezifischen Implementierungen der Interfaces, welche nochmals durch Ordner mit dem Namen der jeweiligen Realisierung getrennt sind. So sind die Hibernate-Umsetzungen unter `hibernate` zu finden. Benötigt eine Implementierung zusätzliche Ressourcen, werden diese ebenfalls in einem extra Ordner untergebracht. Die für Hibernate benötigten Mapping-Dateien sind daher im Verzeichnis `hibernate._hbm` abgelegt.

Im vierten Paket, `de.unileipzig.olate.taskmodel`, befinden sich die Klassen für die Anbindung an das Aufgaben-Framework.

Einen Überblick und genauere Beschreibungen zu den einzelnen Klassen der Pakete bietet die Javadoc-Dokumentation, welche ebenfalls im Rahmen dieser Arbeit erstellt wurde.

5 Implementierung

In diesem Abschnitt wird auf bestimmte Teile der Implementierungsphase eingegangen. Es werden dabei besonders die Bereiche betrachtet, bei denen Probleme aufgetreten und/oder deren Funktions- und Arbeitsweise von besonderer Bedeutung sind. Zudem sollen Entscheidungen, welche bei der Implementierung getroffen wurden, begründet werden.

5.1 ElateTest-Kursbaustein

Wie im Entwurf bereits erläutert, stellt die Olate-Erweiterung einen neuen Kursbaustein „Elate-Test“ zur Verfügung. Dieser ist nach dem einheitlichen Prinzip, mit dem auch die bereits bestehenden Kursbausteine implementiert wurden, realisiert. So wird ein neuer Elate-Test in einen Kurs eingebunden, indem im Kurseditor eine neue Instanz des Bausteins erzeugt wird.

Um mehrere Elate-Test Kursbausteine in einen Kurs einbinden zu können, speichert jede Instanz die Dateien in einem eigenen Ordner. Im Datenverzeichnis von OLAT wird dafür ein Unterordner `elatenodes` im jeweiligen Kursordner erstellt. Dort wiederum wird für jede Instanz ein extra Ordner, mit der ID der Instanz als Name, angelegt. In diesem Ordner befinden sich die XML-Serialisierungen des TaskDefs und der Tasklets der einzelnen Testkandidaten, wobei diese jeweils in einem Unterordner, mit dem Loginnamen als Bezeichner, untergebracht sind.

Wie schon im Abschnitt 4.2.1 erwähnt, gibt es zwei verschiedene Sichten auf einen ElateTest-Kursbaustein. So sehen Kandidaten beispielsweise die Beschreibung zu einem Test, in dem Hinweise zum Test selber enthalten sein können. Diese Beschreibung kann vom Ersteller des Kursbaustein im Kurseditor gesetzt werden. Des Weiteren kann sich hier ein Kandidat, insofern verfügbar, seinen zuletzt durchgeführten und eventuell korrigierten Versuch anschauen. Darunter befinden sich zwei Schaltflächen zum Starten bzw. Fortsetzen eines Versuchs. Die Sicht für die

Abb. 5.1: Die Sicht eines Kandidaten auf einen ElateTest-Kursbaustein.

Korrektoren enthält eine Übersichtstabelle aller Versuche von Kandidaten. Der Status gibt an, in welchem Zustand sich ein Versuch, also ein Tasklet, befindet. Außerdem lässt sich durch einen Klick auf den Kandidaten dessen OLAT-Profil betrachten. Mit der Schaltfläche unterhalb der Tabelle wird der Korrektormodus des Aufgaben-Frameworks gestartet.

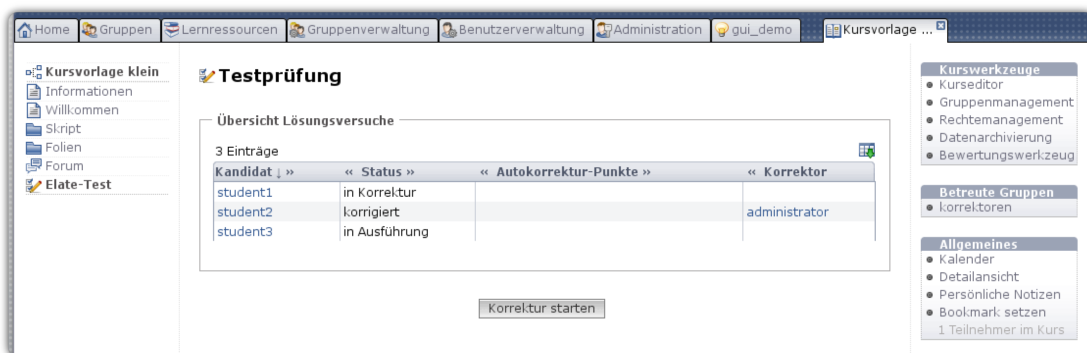

Abb. 5.2: Die Sicht eines Korrektors auf einen ElateTest-Kursbaustein.

5.1.1 Zuordnung der Rollen

Die Zuordnung der beiden Rollen zu den OLAT-Nutzern kann auf verschiedene Weise erfolgen. Korrektoren werden im Kurseditor festgelegt. Dafür werden dem Test eine oder mehrere Lerngruppen zugeordnet, deren Betreuer dann als Korrektoren fungieren. Dadurch lassen sich für jeden Test gezielt die Korrektoren eintei-

len und festlegen. Die Zuweisung eines Korrektors zu einem durchgeführten Test (Tasklet) erfolgt über das Aufgaben-Framework. Testkandidaten sind automatisch alle Nutzer, welche die Berechtigung haben, den Kursbaustein auszuführen, insofern sie nicht Korrektoren sind. Um nur einen bestimmten Nutzerkreis für die Durchführung eines Tests zuzulassen, kann zum Beispiel die Sichtbarkeit und/oder der Zugang zu dem Baustein von einer Gruppe abhängig gemacht werden. Die Mitgliedschaft zu dieser Gruppe kann wiederum durch eine Einschreibung erfolgen. Durch diese Umsetzung können die verschiedensten Szenarien abgebildet werden.

5.2 Integration des Aufgaben-Frameworks

5.2.1 Kommunikation

Da es sich beim Aufgaben-Framework um eine eigenständige Webapplikation handelt, ergeben sich einige Besonderheiten bei der Einbettung in OLAT. Um eine reibungslose Ausführung zu gewährleisten, ohne dass sich beide Anwendungen gegenseitig beeinflussen, stellt OLAT einen speziellen Controller (`IFrameDisplayController` [6]) bereit. Dieser ermöglicht durch Verwendung eines IFrames[1] eine eingebettete Darstellung des Taskmodels direkt in OLAT, ohne mögliche Fehler durch sich überlagerndes CSS, JavaScript usw. Solch ein `IFrameDisplayController` wird in den beiden Sichten zur Darstellung verwendet (Abb. 5.3). Über die Methode `setCurrentURI()` wird eine URL[2] gesetzt, welche das Aufgaben-Framework startet.

Der Austausch von Informationen und Objekten zwischen beiden Anwendungen erfolgt über eine Instanz eines `DelegateObject`, welche im jeweiligen `*RunController` erstellt wird. Diesem *Übertragungsobjekt* wird bei der Erzeugung die ID für das TaskDef, eine Instanz der Klasse `de.thorstenberger.taskmodel.impl.TaskManagerImpl`, welche wiederum die spezifische Implementierung der `TaskFactory` enthält, und der eindeutige Nutzername übergeben. Zusätzlich wird eine URL (*returnURL*) gesetzt, welche später durch das Taskmodel aufgerufen wird. Anschließend wird das Objekt über eine statische Methode (`storeDelegateObjet()`)

[1] zu deutsch „eingebetteter Frame“, ein eigenständiges Gestaltungsmittel als Teil des HTML 4.0 Standards

[2] Uniform Resource Locator, identifiziert und lokalisiert eine Ressource über das verwendete Netzwerkprotokoll

Abb. 5.3: Ausführung des Aufgaben-Frameworks innerhalb von OLAT unter Verwendung eines `IFrameDisplayControllers`.

in der Klasse `de.thorstenberger.taskmodel.TaskModelViewDelegate` gespeichert. Diese ist Bestandteil der Taskmodel-API, welche wiederum als gemeinsam genutzte Klassenbibliothek im Web-Container vorliegt. Das heißt, alle Webanwendungen des Containers nutzen die selben Instanz dieser Bibliothek. Während der Ausführung des Aufgaben-Frameworks wird das dort gespeicherte `DelegateObject` per `getDelegateObject()` geholt und die entsprechenden Informationen werden ausgelesen, unter anderem die zuvor gesetzte *returnURL* und die `TaskManager`-Instanz. Wird ein Test oder eine Korrektur durch den Nutzer beendet, erzeugt das Taskmodel eine HTTP-Anfrage mit der *returnURL*. Diese wird durch OLAT verarbeitet und die entsprechende `event()`-Methode des ausführenden `*RunControllers` aufgerufen. Die Abbildung 5.4 zeigt diesen Ablauf in Form eines Sequenzdiagramms.

5.2.2 TaskFactory

Die wichtigste Tätigkeit bei der Integration des Aufgaben-Frameworks ist die spezifische Implementierung der `TaskFactory`-Schnittstelle durch die Host-Applikation. Dabei müssen die besonderen Gegebenheiten beider Anwendungen berücksichtigt und aufeinander abgestimmt werden. Im Folgenden soll diese Tätigkeit Schritt für Schritt dargestellt werden. Insbesondere wird dabei auf die Probleme eingegangen, welche während der Realisierung aufgetreten sind und wie diese gelöst wurden.

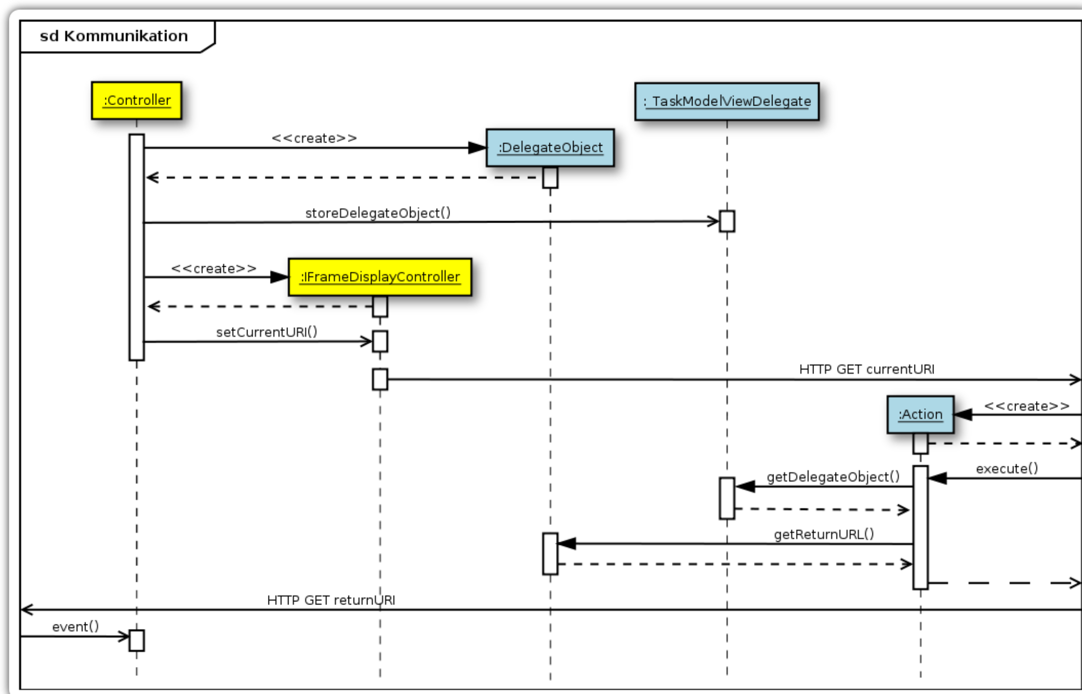

Abb. 5.4: Dieses Sequenzdiagramm zeigt die Kommunikation zwischen den beiden Webapplikationen OLAT (gelbe Objekte) und Aufgaben-Framework (blaue Objekte).

In der Analyse (Abschnitt 3.1.1) wurde bereits dargelegt, dass die `TaskFactory` für die Kapselung sämtlicher Persistenzaufgaben verantwortlich ist. Darüber hinaus bietet sie dem Aufgaben-Framework noch einige weitere Funktionen für die Ausführung. Die wichtigsten dieser Funktionen sollen hier beschrieben werden. Methoden welche keine Erwähnung finden, sind entweder trivial zu realisieren oder spielen für eine Integration im Rahmen dieser Arbeit keine Rolle und werden daher von der Implementierung nicht unterstützt. Sie werfen eine `de.thorstenberger.taskmodel.MethodNotSupportedException`, welche durch das Taskmodel abgefangen wird und signalisiert, dass für diese Methode keine Implementierung vorhanden ist.

Bei der Nutzung von OLAT-eigenen Funktionen innerhalb der `TaskFactory` muss darauf geachtet werden, dass diese keine Datenbankoperationen hervorrufen. Da das Aufgaben-Framework und somit auch die `TaskFactory`-Instanz in einem eigenen Kontext ablaufen, kann dies zu Fehlern in der Verwaltung der Datenbanksitzung von OLAT führen. Im Punkt Persistenzfunktionen wird noch einmal genauer auf dieses Problem eingegangen.

Instanziierung

Bei der Erzeugung einer Instanz dieser Schnittstelle müssen sämtliche Informationen übergeben werden, die für die Ausführung der verschiendenen Methoden nötig sind. Dies sind im wesentlichen Objektkopien, mit denen die `TaskFactory` arbeitet. Gesondert zu nennen wären hier die beiden DAOs für den Zugriff auf die TaskDefs und Tasklets, der Pfad zum Kursordner, sowie die ID und eine Liste aller Korrektoren des Kursbausteins.

Persistenzfunktionen

Die Persistenzfunktionen der `TaskFactory` laden bzw. speichern die von der Hostanwendung verwalteten Datenobjekte und müssen gegebenfalls Informationen zwischen den verschiedenen Formaten transformieren.

Insgesamt gibt es drei Methoden zum Laden von TaskDefs (`getTaskDef*()`). Zwei davon laden mehrere TaskDefs gleichzeitig. Da die Integration durch Olate mit Hilfe eines Kursbausteins erfolgt und jeweils immer nur ein TaskDef zugeordnet ist, musste auch nur die Methode `getTaskDef(long key)` implementiert werden. Diese lädt anhand des zugehörigen Data Access Objects und dem übergebenen Schlüssel das entsprechende Datenobjekt eines TaskDefs aus der OLAT-Datenbank und erzeugt mit diesen Informationen eine `TaskDef_ComplexImpl`-Instanz, mit der das Framework arbeitet. Da die Verwaltung der Tests komplett über OLAT geschieht, ist es nicht notwendig, dass TaskDefs durch das Taskmodel gespeichert bzw. gelöscht werden können. Aus diesem Grund werden die beiden Methoden `storeTaskDef()` und `deleteTaskDef()` nicht unterstützt, sie werfen also eine `MethodNotSupportedException`.

Wie in der Analyse bereits dargestellt, steuert das Aufgaben-Framework den kompletten Lebenzyklus eines Tasklets. So auch das Erzeugen einer neuen Instanz, wenn ein Testkandidat zum ersten Mal einen Test durchführt. Dies erfolgt über die Funktion `createTasklet(String userId, long taskId)`. Sie erstellt eine neue Tasklet-Datenobjekt-Instanz (`TaskletVo`), setzt alle relevanten Informationen und speichert diese initial in der Datenbank. Anschließend wird ein `Tasklet` instanziiert. Hat ein Testkandidat bereits an einem Test teilgenommen, so wird dessen frühere Durchführung durch die Methode `getTasklet(String userId, long taskId)` geladen. Da auch das Taskmodel nach jeder HTTP-Anfrage, zum Beispiel das Speichern einer Antwort oder die Abgabe eines Tests, den aktuellen Zustand

persistiert, muss die Funktion `storeTasklet(Tasklet tasklet)` realisiert werden. Hier müssen alle Informationen, welche nicht in der XML-Persistierung gespeichert werden, in das Datenobjekt übertragen werden. Aus Performanzgründen werden nur geänderte Daten aktualisiert. Um den Korrekturmodus zu ermöglichen, werden über die Methode `getTasklets(long taskId)` alle zu einem Test existierenden Tasklets geladen. Wie schon bei den TaskDefs werden die Tasklets über OLAT verwaltet. Somit wird das Entfernen eines Tasklets über `removeTasklet()` nicht unterstützt.

Bei der Realisierung der Persistenzfunktionen traten anfangs einige Fehler auf. Da die Data Access Objects zwar im OLAT-Kontext instanziiert werden, deren Nutzung aber im Kontext des Aufgaben-Frameworks stattfindet, gab es Probleme mit der Datenbanksitzung von OLAT, welches für die Hibernate-Einbindung in der Persistenzschicht das „Sitzung-je-Anfrage“-Muster [7] anwendet. Das heißt, mit jeder HTTP-Anfrage des Nutzers wird eine neue Hibernate-Sitzung eröffnet und am Ende dieser Anfrage wieder geschlossen. Durch die Einbettung des Taskmodels wird dieser Ablauf jedoch gestört, was sich durch ungültige Sitzungen oder Zeitüberschreitungen bemerkbar macht. Dieses Problem konnte durch die Verwendung von JNDI[3] gelöst werden. Mit Hilfe von JNDI wird die OLAT-eigene `SessionFactory`, welche zum Erzeugen von Sitzungen genutzt wird, über eine eindeutige Kennung im Namensraum des Web-Containers gebunden. Diese kann dann im Kontext des Aufgaben-Frameworks geladen und verwendet werden, um eigene Sitzungen zu erzeugen. Sämtliche Datenbankoperationen werden über das Anti-Muster „Sitzung-je-Operation“ durchgeführt. Laut Hibernate-Dokumentation sollte die Verwendung dieses Musters möglichst vermieden werden. Da aber der Ablauf, in dem die Methoden der `TaskFactory` aufgerufen werden, sich nicht beeinflussen bzw. kontrollieren lässt, ist eine Anwendung anderer Muster nicht möglich. So ist dies ein seltener Fall, in dem dieses Anti-Muster auf Grund der Gegebenheiten angewandt werden muss, um gewährleisten zu können, dass sämtliche Datenbankoperationen fehlerfrei durchgeführt werden.

Weitere Funktionen

Mit Hilfe der Methode `availableTypes()` wird angegeben, welche Aufgabentypen durch diese Implementierung zur Verfügung stehen. Insgesamt unterstützt das Aufgaben-Framework drei verschiedene Typen, `COMPLEX`, `REMOTE` und `UPLOAD`.

[3]Java Naming and Directory Interface - `http://java.sun.com/products/jndi/`

Der wichtigste und für Prüfungsaufgaben relevante Typ ist eine `COMPLEX`-Aufgabe, welche wiederum aus verschiedenen Teilaufgaben besteht. Da dieser durch die Olate-Erweiterung unterstützt werden soll, muss die durch die genannte Methode zurückgegebene Liste nur diesen speziellen Typ enthalten.

Wie bereits erwähnt, werden die Korrekturen auch über das Aufgaben-Framework abgewickelt. Darüber hinaus ist es ebenfalls für die Zuordnung von Korrektoren zu einem Tasklet verantwortlich. Über die Methode `getCorrectors()` erhält es eine Liste aller Korrektoren. Durch die Möglichkeit für jeden Kursbaustein und somit Test die Korrektoren gesondert festzulegen, ist es an dieser Stelle notwendig, die genaue Test-Instanz zu kennen. Jedoch besitzt die Funktion keinerlei Parameter zum Ermitteln dieser. Da aber die `TaskFactory` erst bei Betreten des Kursbausteins erzeugt wird, steht zu diesem Zeitpunkt fest, um welche Instanz es sich handelt. Somit kann diese Information dem Konstruktor übergeben und als Klassenvariable gespeichert werden. Diese wird dann beim Erstellen der Korrektoren-Liste verwendet.

5.2.3 Vorschaufunktion

Der Kurseditor von OLAT besitzt eine Funktion, mit der man einen Kurs in einer Vorschau betrachten und testen kann, ohne diesen vorher publizieren zu müssen (siehe [1]: Abschnitt 9.3.2). Die Implementierung der Olate-Erweiterung unterstützt ebenfalls diese Vorschaufunktion. Jedoch ist diese auf Grund von technischen Gegebenheiten eingeschränkt. Da die Vorschau im Editormodus durchgeführt wird, stellt die Kurslaufzeitumgebung einige Funktionen, welche im Normalbetrieb benötigt werden, nicht zur Verfügung. Somit kann nicht bestimmt werden, ob ein Nutzer sich in der Lerngruppe der Korrektoren befindet. Deshalb bekommt man in der Vorschau immer die Sicht eines Testkandidaten angezeigt.

Eine Vorschau bedeutet, dass keinerlei Daten über die Dauer dieser Vorschau hinaus persistiert werden müssen. Daher wurde in die `TaskFactory`-Implementierung ein Vorschaumodus integriert, der verhindert, dass schreibende Datenoperationen ausgeführt werden. Dies ließ sich aber nicht an allen Stellen realisieren. So nutzt das Taskmodel FileInputStreams zum Einlesen von Daten, welche aber nur auf existierenden Dateien arbeiten können. Das würde aber wiederum den realen Testbetrieb stören bzw. sogar beeinflussen. Aus diesem Grund wird für den Vorschaumodus das temporäre Verzeichnis des Betriebssystems genutzt, welches

über die JVM[4] mit Hilfe der System-Klasse ermittelt werden kann (`System.getProperty('java.io.tmpdir')`). In diesem Verzeichnis werden sämtliche für die Vorschau benötigten Dateien angelegt und gespeichert. Zusätzlich besitzt das Taskmodel noch einen Cache, welcher die Tasklets speichert und somit Leseoperationen vermindert. Die Referenzimplementierung des `TaskletContainer` aus dem Taskmodel bietet aber keine Möglichkeit, die Tasklets manuell aus dem Cache zu löschen. Durch den Austausch gegen eine eigene Umsetzung (`TaskletContainerOlatImpl`) wurde diese Funktion hinzugefügt. Nach Beendigung der Vorschau wird das dafür erstellte `Tasklet`-Objekt wieder aus diesem Cache entfernt. Mit Hilfe dieser Maßnahmen wird gewährleistet, dass die Daten des Realbetriebes nicht beeinflusst werden.

[4]Java Virtual Machine, ist für die Ausführung des Java-Bytecodes verantwortlich

6 Versionsmanagement

Software-Produkte unterliegen im Allgemeinen einem ständigen Evolutionsprozess. Es werden neue Versionen mit neuen Funktionen oder wichtigen Fehlerkorrekturen veröffentlicht. Unter Umständen kann dies bedeuten, dass sich bestimmte Schnittstellen ändern, worauf Anwendungen, welche diese nutzen, angepasst werden müssen. Aber auch Anforderungen derer, die eine Software einsetzen, können sich ändern, wodurch ebenfalls eine Anpassung des Produkts durchgeführt werden muss.

Auch das Olate-Projekt unterliegt einem solchen Prozess. So ist es sehr wahrscheinlich, dass die erste Version, welche im Rahmen dieser Arbeit entstanden ist, durch den produktiven Einsatz etwaige Mängel bzw. Fehler aufzeigen wird und neue Anforderungen an die Funktionalität entstehen. Des Weiteren verbindet Olate zwei voneinander unabhängige Anwendungen, welche sich fortwährend weiterentwickeln und regelmäßig in neuen Versionen erscheinen. Somit muss, je nach Bedarf, auch Olate selbst an diese neuen Versionen angepasst werden. Im Folgenden soll auf diese Problematik eingegangen und Richtlinien sowie das Vorgehen bei Versionänderungen vorgestellt werden.

6.1 Allgemeine Richtlinien

Bei der Entwicklung des Gesamtmodells wurden bestimmte Technologien und Muster eingesetzt. Diese sollten bei einer weiteren Entwicklung angewandt bzw. erweitert werden. Die Einhaltung solcher Vorgaben gewährleistet dabei, dass nicht jedes neue Feature seinen eigenen Entwurf und/oder Designentscheidungen mit sich bringt. Bei der Realisierung neuer Komponenten oder Features sollte immer überprüft werden, ob das Gesamtmodell bereits Vorgaben und Muster durch ähnliche Features bereitstellt. Im positiven Falle sollten diese verwendet werden, sonst

ist eine Implementierung unter Anwendung weiterer standardisierter Muster vorzuziehen.

Als Beispiel wären hier das Datenmodell und die Persistierung zu nennen. Beide Komponenten wurden komplett gegen Interfaces erstellt und eine Standardimplementierung vorgegeben. Mit Hilfe der `DaoFactory` wird diese Implementierung zur Verfügung gestellt. Wird in der Zukunft eine andere Umsetzung benötigt, kann diese den Komponenten hinzugefügt werden. Anschließend müssen lediglich die beiden Methoden der `DaoFactory` angepasst werden. Um dieses Prinzip zu unterstützen, muss natürlich gewährleistet werden, dass die DAO-Instanzen immer über die Factory erzeugt werden.

Eine weitere wichtige Vorgabe ist die Bereitstellung von Olate als Erweiterung, ohne dass der OLAT-eigene Quellcode verändert werden muss. Dadurch wird eine bestmögliche Kompatibilität und einfache Einsetzbarkeit erreicht. So lässt sich Olate ohne großen Aufwand installieren und auch wieder deinstallieren. Dieses Ziel sollte auch in zukünftigen Entwicklungen berücksichtigt werden.

6.1.1 Konfiguration und Versionierung

Bei der Entwicklung werden die Vorgaben aus dem Konfigurationsmanagement der Abteilung Betriebliche Informationssysteme [5] umgesetzt. Die Arbeiten am Olate-Projekt finden somit als eigenständige Konfigurationseinheit im Branch der Konfigurationsdatenbank statt. Die verschiedenen Versionen werden im Tags-Ordner des SVN[1]-Repositories verwaltet. Die Versionierung erfolgt grundsätzlich durch Versionsnummern. Dabei wird das folgende Schema angewandt:

OLATE_[M.F.B]_[YYYYMMDD]

Die Versionsnummer setzt sich aus drei natürlichen Zahlen und dem Veröffentlichungsdatum zusammen. Die erste Zahl steht für ein Major-Release, also eine Version, welche fundamentale Änderungen bzw. Neuerungen erfahren hat. Die zweite Zahl gibt Feature-Releases an. Diese entstehen durch neue Anforderungen, welche aber ohne aufwendige Änderungen an der Quellcode-Basis implementiert werden können. Die letzte und dritte Ziffer bezeichnet Bugfix-Releases. Hier werden lediglich entdeckte Fehler beseitigt.

[1] Abkürzung für Subversion, eine Software zur Versionsverwaltung von Dateien und Verzeichnissen

6.1.2 Bereitstellung

Der Quellcode jeder Version kann über die Konfigurationsdatenbank bezogen werden. Da Olate aber eine Erweiterung für die OLAT-Plattform darstellt, ergeben sich hierfür weitere Vorgaben. So wird Olate, wie bereits in Abschnitt 3.2.1 erläutert, als Java-Archiv ausgeliefert und über eine Konfigurationsdatei in OLAT eingebunden. Des Weiteren nutzt jedes Release eine bestimmte Version des Aufgaben-Frameworks, welches wiederum aus drei Teilen besteht, der Taskmodel-API, dem Taskmodel-CORE und der Webapplikation Taskmodel-CORE-View. Diese werden jedem Release-Bündel beigefügt. Als letztes muss eine für die jeweilige Version angepasste Installationsanleitung bereitgestellt werden. All diese genannten Bestandteile sollten zusammen in Form eines geeigneten Bündels über eine zentrale Stelle zur Verfügung gestellt werden. Die aktuelle Version, welche während dieser Arbeit entstandt, kann über den Ordner *tags/OLATE_1.0.0_20090907* aus der Konfigurationsdatenbank als Quellcode oder fertiges Paket bezogen werden.

6.2 Releasemanagement

Wie bereits schon erwähnt muss die Olate-Erweiterung regelmäßig an neue Anforderungen und Versionen der verwendeten Applikationen angepasst werden. Hier eignet sich eine Anlehnung an die Release-Zyklen der beiden Fremdprojekte. Für das Aufgaben-Framework besteht jedoch zur Zeit kein fester Releaseplan, womit sich in diesem Falle an den Plan des OLAT-Projekts gehalten werden sollte. Es eignet sich auch hier, wie im Konfigurationsmanagement der Abteilung Betriebliche Informationssysteme vorgeschlagen, den Prozess für eine Weiterentwicklung drei Monate nach einem neuen Major- oder Feature-Release von OLAT anzustoßen. Dadurch sollte als Grundlage für eine Anpassung eine durch Bugfix-Releases weitgehend fehlerfreie OLAT-Version dienen. Zu diesem Zeitpunkt sollte auch geprüft werden, ob sich ein Aktualisierung auf eine neue Taskmodel-Version lohnt. Wenn dies der Fall ist, muss eine feste SVN-Revision für das Taskmodel festgelegt werden, welche im weiteren Verlauf der Weiterentwicklung von Olate verwendet wird.

6.2.1 Vorgehen

Das Vorgehen bei der Realisierung einer neuen Version richtet sich weitestgehend nach dem festgelegten Entwicklungsmodell (siehe Abschnitt 2.2). Es werden initial die ersten drei Prozesse nochmals durchlaufen, wobei die bisher vorliegenden Dokumente des jeweiligen Prozesses entsprechend den neuen Anforderungen erweitert bzw. angepasst werden. Dabei ist es von Vorteil, neue Features mit der Versionsnummer des jeweiligen Olate-Releases zu versehen, um besser nachvollziehen zu können, in welcher Version welches Feature neu hinzukamen. Anschließend werden die beiden Prozesse vier und fünf wieder jeweils zusammen pro Feature ausgeführt.

Auf der technischen Seite bedeutet ein neues Release, dass die beiden Fremdprojekte in der verwendeteten Entwicklungsumgebung, zum Beispiel Eclipse[2], aktualisiert werden. Daraufhin sollten bei Veränderungen der API der Projekte durch die Umgebung Fehler im Olate-Projekt angezeigt werden. Diese müssen nun zuerst behoben werden. Zusätzlich sollte durch geeignete Tests geprüft werden, ob sich eventuell das Verhalten der Erweiterung mit den neuen Versionen geändert hat. Auch hier müssen dann entsprechende Änderungen durchgeführt werden. Erst nach diesen Tätigkeiten sollte mit der Einbindung und Implementierung der neuen Features begonnen werden, um sicherstellen zu können, dass die verwendete Basis weitgehend fehlerfrei funktioniert.

Sind alle Anpassungs- und Erweiterungsarbeiten abgeschlossen, muss eine neue Version in der Konfigurationsdatenbank im Ordner *tags* angelegt werden. Außerdem muss jede Version eine Datei *README.txt* enthalten, in der die für dieses Release verwendeten Versionen der beiden Fremdprojekte, sowie weitere Hinweise enthalten sind. Auch die Installationsanleitung muss eventuell angepasst werden. Abschließend muss ein neues Release-Bündel erzeugt und bereitgestellt werden.

6.3 Fehlerverwaltung

Bei der Entwicklung des Olate-Projekts wurden zwei Fehler in der verwendeten OLAT-Version 6.1 entdeckt.

[2]quelloffenes Programmierwerkzeug zur Entwicklung von Software

Zum einen gibt es das Problem, dass der Aufruf zum Entfernen nicht mehr benötigter Daten beim Löschvorgang eines Kursbausteins nur dann stattfindet, wenn dieser schon einmal publiziert wurde. Das heißt, wenn ein neuer Kursbaustein erstellt, aber niemals publiziert wird, werden eventuell gespeicherte Daten (Dateien, Datenbankeinträge) beim Löschen des Bausteins nicht entfernt. Dies führt zu einem Anstieg des Platzbedarfs durch unnötige Dateien und zu nicht mehr gültigen Datenbankeinträgen. Eine Auswirkung auf die Stabilität des Systems hat dies aber nicht.

Ein zweiter Fehler besteht in der Vorschau des Kurseditors. Klickt man in der Vorschau durch die einzelnen Kursbausteine, so wird bei jedem Klick automatisch durch OLAT eine `dispose()`-Methode des jeweiligen `Controllers` aufgerufen, die dazu verwendet werden kann, um bestimmte Tätigkeiten beim Verlassen/Schließen eines Kursbausteins durchzuführen. Wird jedoch die Vorschau direkt über „Vorschau beenden“ geschlossen, gleich nachdem man einen Kursbaustein angewählt hat, so wird diese entsprechende Methode nicht aufgerufen. Im Falle des ElateTest-Kursbausteins heißt das, dass der Tasklet-Cache (siehe Abschnitt 5.2.3) nicht korrekt geleert wird und somit der Realbetrieb gestört werden könnte. Da aber in der Praxis nur die Besitzer eines Kurses auch dessen Vorschau betrachten können, diese aber im Normalfall nicht selbst als Kandidaten an einem Test teilnehmen, sollte es zu keinen Problemen kommen.

Gefundene Fehler sollten dem jeweiligen Fremdprojekt mitgeteilt werden. So wurden die beiden genannten Fehler an das Bugtracking-System von OLAT gemeldet und sind unter folgenden URLs einsehbar:

- `http://bugs.olat.org/jira/browse/OLAT-4017`
- `http://bugs.olat.org/jira/browse/OLAT-4017`

Es empfiehlt sich, die eingesandten Fehler zu abonnieren, um bei Änderungen automatisch informiert zu werden.

Des Weiteren wurden die entsprechenden Stellen per Javadoc im Quellcode von Olate markiert. Dabei wurde folgendes Schema, welches auch bei zukünftig entdeckten Fehlern angewandt werden sollte, genutzt:

> **TODO:** Bug: <OLAT Issue-Key>, <Beschreibung>

Ist ein solcher Fehler nicht geeignet umgehbar, muss entweder eine andere Realisierung der Funktion umgesetzt werden oder der Fehler wird selbst beseitigt und

ein entsprechender Patch an das Entwicklerteam weitergereicht. Letzteres sollte aber nur im äußersten Notfall durchgeführt werden, weil dadurch das Ziel, den Quellcode der Fremdprojekte nicht zu verändern, verhindert wird.

7 Zusammenfassung

Die vorliegende Arbeit verfolgte die Beantwortung der im Abschnitt 1.1 gestellten Fragestellungen.

So wurde nach ausführlicher Diskussion Feature-Driven-Development als geeignetes Vorgehensmodell ausgewählt. Dieses hat sich in der Praxis als sehr komfortabel und leicht anwendbar bewiesen. Die verschiedenen Komponenten wurden in feingranulare Features aufgeteilt. Danach wurden ähnliche Features herausgearbeitet und bei der Umsetzung nach gleichen Mustern implementiert. Durch die Aufteilung in einzelne, unabhängig zu erstellende, Features können die anfallenden Implementierungsarbeiten ohne großen Aufwand einfach unter den verschiedenen Entwicklern aufgeteilt werden. Ein weiterer Vorteil besteht darin, dass relativ schnell ein erster lauffähiger Prototyp entsteht, der schon während der Entwicklung in Reviews zwischen den Entwicklern selbst und mit dem Auftraggeber vorgezeigt und als „anfassbares“ Diskussionsmaterial verwendet werden konnte.

Von Anfang an bestand das Ziel, die Quellcode-Basis von OLAT und dem Aufgaben-Framework bei einer Integration unberührt zu lassen. Nach einer Analyse beider Applikationen stellte sich somit die Integration durch einen neuen Kursbaustein als die beste Lösung heraus. Dadurch konnte bereits bestehende Funktionalität genutzt werden. Insbesondere das Gruppenmanagement eines Kurses wird in einem ElateTest-Kursbaustein dazu verwendet, um die Korrektoren eines Tests zu definieren. Die Ausführung eines Tests erfolgt innerhalb der OLAT-Umgebung. Dies wird durch die Verwendung einer speziellen Komponente realisiert, welche die Darstellung einer externen Anwendung direkt in OLAT ermöglicht. Die Anbindung des Aufgaben-Frameworks erfolgte durch die Implementierung einer dafür geschaffenen Schnittstelle. Dabei sind einige Probleme aufgetreten, die jeweils zusammen mit einer konkreten Lösung erläutert wurden. Für die Persistierung der anfallenden Daten wurde eine zweigleisige Realisierung gewählt. Die TaskDefs und Tasklets werden als XML-Dateien im OLAT-Datenverzeichnis gespeichert, Meta-Informationen dagegen in der OLAT-Datenbank.

Im letzten Teil der Arbeit ging es um die Beantwortung der Frage, wie eine zukünftige Weiterentwicklung aussehen könnte. Es wurde die Verwaltung des Olate-Projekts als Einheit in der Konfigurationsdatenbank des Lehrstuhls für Betriebliche Informationssysteme beschrieben. Dies beinhaltet ein Schema zu Versionierung und das Vorgehen bei Anpassungen an neue Versionen der beiden integrierten Anwendungen bzw. beim Einarbeiten von neuen Anforderungen. Außerdem wurden zwei Fehler vorgestellt, welche während der Implementierungsarbeiten aufgefallen sind und wie in solchen Situationen vorzugehen ist.

7.1 Ausblick

Bei der Verwendung des `IFrameDisplayControllers` zur Darstellung des Aufgaben-Frameworks innerhalb der OLAT-Umgebung ergeben sich folgende Nachteile. So kann es in bestimmten Situationen dazu kommen, dass ein zweiter vertikaler Scrollbalken erscheint, was eine schnelle und komfortable Navigation verhindert, was besonders in zeitkritischen Prüfungssituationen negativ auffällt. Des Weiteren besitzt die Webapplikation Taskmodel-CORE-VIEW als Teil des Aufgaben-Frameworks ein andere Optik und Haptik als OLAT selbst. Diese Schwachstellen könnten zum Beispiel durch einen Austausch der View-Komponente des Taskmodels beseitigt werden. So wäre eine Implementierung der Testausführung, ähnlich der der IMS-QTI[1]-Tests, direkt über OLAT denkbar. Durch eine solche Lösung würde sich auch der Installations- und Einrichtungsaufwand deutlich verringern.

[1] IMS Question & Test Interoperability, ein standardisiertes Datenformat zur Konzipierung austauschbarer Tests und Quizzes

Abbildungsverzeichnis

Abb. 2.1: FDD-Prozessmodell . . . 12
Abb. 3.1: Tasklet-Lebenszyklus . . . 14
Abb. 4.1: Klassendiagramm OLAT-Erweiterung . . . 18
Abb. 4.2: Klassendiagramm Datenmodell . . . 20
Abb. 4.3: Klassendiagramm Persistierung . . . 22
Abb. 5.1: Kandidaten-Sicht des ElateTest-Kursbausteins . . . 26
Abb. 5.2: Korrektoren-Sicht des ElateTest-Kursbausteins . . . 26
Abb. 5.3: Aufgaben-Framework in Ausführung . . . 28
Abb. 5.4: Kommunikation OLAT – Aufgaben-Framework . . . 29

Literaturverzeichnis

[1] *OLAT 6 - Benutzerhandbuch.* http://www.olat.org/website/en/download/help/OLAT_6_1_Manual_DE_print_090306.pdf, 3.2009 v6.1 Aufl.

[2] *OLAT 6 - Funktionsübersicht.* http://www.olat.org/website/en/download/OLAT_6_0_Funktionsuebersicht.pdf, 1.2009 v4.0 Aufl.

[3] Beck, K.: *Extreme Programming - Das Manifest.* Addison-Wesley Verlag, 2000.

[4] Berger, T. und H.-W. Wollersheim: *Eine dienste- und komponentenbasierte Architektur zur elektronischen Durchführung von Prüfungen und zum Management von Lehrveranstaltungen.* GI-Edition - Lecture Notes in Informatics (LNI), 2006.

[5] Gerber, D.: *Operative Prozessunterstützung durch Integration einer Prüfungsverwaltung in das LMS-OLAT.* Bachelorarbeit, Abt. Betriebliche Informationssysteme, Institut für Informatik, Universität Leipzig, 2009.

[6] Gnägi, F.: *org.olat.core.gui.control.generic.iframe (OLAT Javadoc / API documentation).* www.olat.org.

[7] King, G., C. Bauer, M. R. Andersen, E. Bernard und S. Ebersole: *Hibernate Reference Documentation.* http://www.hibernate.org/hib_docs/v3/reference/en-US/pdf/hibernate_reference.pdf, 3.3.2.GA Aufl.

[8] Luca, J. D.: *The Latest FDD Processes.* http://www.nebulon.com/articles/fdd/latestfdd.html, v1.3 Aufl.

[9] Sun Microsystems, Inc.: *Core J2EE Patterns - Data Access Object.* http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html.

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, dass alle Stellen der Arbeit, die wörtlich oder sinngemäß aus anderen Quellen übernommen wurden, als solche kenntlich gemacht sind und dass die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegt wurde.

Leipzig, den 7. September 2009

Marvin Frommhold