Hans Pongratz, Reinhard Keil (Hrsg.): Die 13. E-Learning Fachtagung Informatik,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2015 169

Grappa – die Spinne im Netz der Autobewerter und Lernmanagementsysteme

Robert Garmann[1], Felix Heine[2] und Peter Werner[3]

Abstract: Wie können wir es schaffen, unter hohen Aufwänden entstandene, automatisiert bewertbare Programmieraufgaben auf vielen verschiedenen Lernplattformen zur Verfügung zu stellen? Wie muss eine „Verteil"-Software beschaffen sein, um Studierenden und Lehrenden, die häufig nur ein einziges Lernmanagementsystem nutzen (wollen), Zugang zu verschiedensten Autobewertern zu verschaffen? Dieser Beitrag stellt Anforderungen, Entwurfskonzepte und einige technische Details der Softwarekomponente „Grappa" vor - einer an der Hochschule Hannover entwickelten und im Einsatz befindlichen Middleware, die auf die Anbindung verschiedener Autobewerter an verschiedene E-Learning-Frontends spezialisiert ist.

Keywords: E-Assessment, Programmieraufgabe, Middleware, Grader, Plugin.

1 Einleitung

Unter Programmieraufgaben in der Informatikausbildung verstehen wir ein weites Spektrum. Dieses reicht von klassischen Programmieraufgaben in Allzwecksprachen wie Java über Aufgaben in spezialisierten Abfragesprachen wie SQL bis hin zu Modellierungsaufgaben in abstrakten Notationen wie der UML. So verschieden die verwendeten Sprachen sind, so verschieden sind die zur automatisierten Bewertung studentischer Lösungen eingesetzten Autobewerter („Grader").

Auf der anderen Seite nutzen Lehrende und Studierende verschiedenste Lernmanagementsysteme (LMS) für die Bereitstellung und Konsumierung von Inhalten. Idealerweise nutzen Studierende dasselbe LMS zur Einreichung von Lösungen zu Aufgaben und zum Abrufen des automatisch generierten Feedbacks eines Graders. Viele LMS sind individuell erweiterbar. So ist es möglich, mit Hilfe von extra entwickelten „Plugins" bzw. „Adaptern" fast jeden denkbaren Grader, der dafür vorbereitet ist, an jedes LMS anzubinden (vgl. Abb. 1), allerdings um den Preis vieler spezieller und separat zu wartender Adapter.

Die in diesem Beitrag[4] vorgestellte Softwarekomponente „Grappa" will größtmögliche

[1] Hochschule Hannover, Fakultät IV Wirtschaft und Informatik, Ricklinger Stadtweg 120, 30459 Hannover, robert.garmann@hs-hannover.de
[2] Hochschule Hannover, Fakultät IV Wirtschaft und Informatik, Ricklinger Stadtweg 120, 30459 Hannover, felix.heine@hs-hannover.de
[3] Hochschule Hannover, ZSW - E-Learning Center, Expo Plaza 12, 30539 Hannover, peter.werner@hs-hannover.de
[4] Dieser Beitrag wurde als Teil des Projekts „eCompetence and Utilities for Learners and Teachers" (eCULT)

Vernetzbarkeit von LMS und Gradern bei gleichzeitig geringer Anzahl und Komplexität der Adapter erreichen. Gleichzeitig wird durch Grappa eine homogene Bedienung der unterschiedlichen Grader innerhalb eines LMS erreicht. Wir erläutern Anforderungen und Architektur sowie daraus resultierende Entwurfsentscheidungen und einige technische Details. Grappa wird derzeit an der Hochschule Hannover in Programmieren-Lehrveranstaltungen zusammen mit moodle[5] und aSQLg [KTH13] eingesetzt. Ein Einsatz mit Graja [Gar13] ist in kleinem Rahmen erfolgt, die Anbindung weiterer LMS (ppkm[6], LONCAPA[7]) ist geplant. Eine Evaluation des bisherigen Einsatzes ist in [SBG+14] zu finden.

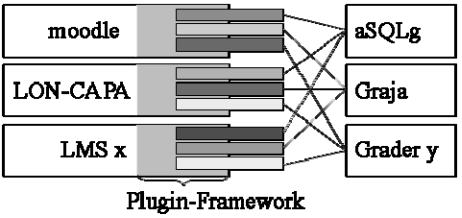


Abb. 1: Individuelle Anbindung von Gradern an LMSe. An jeder Linie wird eine speziell angefertigte Softwarekomponente (dargestellt als verschiedenfarbige Kästchen) entwickelt und gepflegt – abgestimmt auf das Plugin-Framework des LMS (grau schattiert dargestellt).

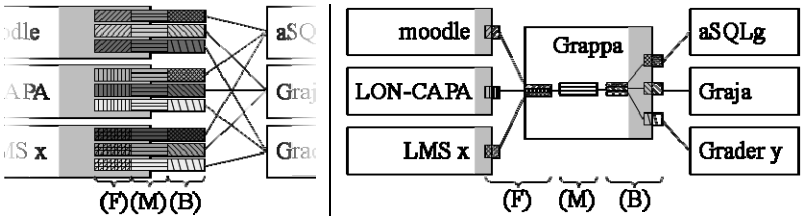
2 Architekturidee

Ideal wäre sicher, ein universelles Austauschformat für Programmieraufgaben zu haben, welches zur Kommunikation und zum Datenaustausch zwischen allen beteiligten LMSen und Autobewertern genutzt werden könnte. Ein solches Austauschformat reduziert die Aufwände für Adapterentwicklungen, eliminiert sie jedoch nicht. Häufig sind es technische Details wie Zeichenkodierungen o. ä., die die Anbindung eines Graders an ein LMS kompliziert machen. Zudem besitzt jedes LMS und jeder Grader technische Besonderheiten, die bei der Adapterentwicklung aufwändig und individuell berücksichtigt werden müssen. Ein Grader spricht vielleicht REST, der nächste SOAP, ein weiterer ist evtl. nur als Shellskript aufrufbar. Das eine LMS akzeptiert nur in php programmierte Plugins, das andere nur in Java vorliegende. Eine Zusammenführung der Grader Backends hinter einer gemeinsamen Fassade würde die Entwicklungs- und Wartungsaufwände auf Seiten der LMSe minimieren.

Abb. 2b illustriert diese Idee. Grappa agiert als Spinne im Netz zwischen LMSen und Gradern. Jeder für ein LMS-Grader-Paar spezifische Adapter wird intern als dreigeteilt

vom Bundesministerium für Bildung und Forschung (BMBF) gefördert (Förderkennzeichen 01PL11066D).
[5] https://moodle.org/
[6] Eigenentwicklung der HS Hannover
[7] http://www.lon-capa.org/

aufgefasst (Abb. 2a). Einige Funktionen sind LMS-spezifisch (F=frontend), einige Grader-spezifisch (B=backend) und einige sind in ähnlicher Form in jedem Adapter vorhanden (M=mediator). Grappa übernimmt die Funktion (M) und einige Anteile der Funktionen (F) und (B).



(a) (b)

Abb. 2: (a) Jede der in Abb. 1 dargestellten Komponenten vereint konzeptionell drei Aufgabenbereiche (F/M/B – vgl. Erläuterung im Text). (b) Vereinfachte Architektur mit Grappa, bei der ähnliche oder gleiche Aufgaben an zentraler Stelle umgesetzt sind. LMS- bzw. Grader-spezifische Funktionen werden als LMS-Plugin bzw. als Grappa-BackendPlugin realisiert (grau schattiert) und sind von geringer Komplexität.

LMS-Plugins stützen sich auf von Grappa in verschiedenen Programmiersprachen angebotene Clientbibliotheken (F) und sind dadurch sehr schlank. Ein Beispiel einer (F)-Funktion ist die asynchrone Ansteuerung des Bewertungsprozesses.

Grader können in ein von Grappa angebotenes Backend-Plugin-Framework (B) integriert und so ebenfalls schlank gestaltet werden. Ein Beispiel für eine (B)-Funktion ist die technische Anbindung des Graders, z. B. der Aufruf eines Shell-Skripts. Ein weiteres Beispiel ist die Umwandlung von textuellem Feedback des Graders in eine präsentationsfähige Form.

Grappa (M) übernimmt neben der technischen Anbindung weitere LMS-übergreifende Funktionen wie das Queueing von Anfragen und die Umwandlung von Bewertungsskalen. Mit einem eigenen Datenbestand versehen kann Grappa zudem den Austausch von Programmieraufgaben über LMS-Grenzen hinweg bewerkstelligen.

3 Anforderungen

Die zentrale Funktion der Grappa-Komponente besteht darin, eine studentische Lösung zu einer Aufgabe vom LMS entgegenzunehmen, durch einen passenden Grader bewerten zu lassen und das Ergebnis an das LMS zurückzuliefern. Da Grappa unabhängig von einem konkreten LMS oder Grader ist, müssen verschiedene Anforderungen sowohl seitens der LMS als auch seitens der Grader berücksichtigt werden. Insbesondere darf die Art der Aufgabe (Programmieraufgabe Java, Datenbankaufgabe SQL, Modellierungsaufgaben, etc.) keine Rolle spielen. Diese Anforderungen leiten sich aus den unterschiedlichen Fähigkeiten der Grader sowie der LMSe ab.

Die **Einreichungen** können vom Studierenden im LMS entweder als Dateien bzw. Archiv hochgeladen oder über ein Texteingabefeld direkt eingeben werden. Unabhängig davon sendet das LMS immer eine oder mehrere Dateien an Grappa. Diese Dateien werden von Grappa nicht analysiert. Sie werden lediglich unverändert an das Grader-BackendPlugin weitergeleitet.

Eine Aufgabe kann aus Sicht des LMS in mehrere Unteraufgaben oder Teilaspekte wie Syntax oder Semantik aufgeteilt sein. Diese Struktur spiegelt sich in der Bewertung der Aufgabe wider. Der Grader kann dieser Struktur noch weitere Ebenen hinzufügen, indem er zu einer Teilaufgabe bestimmte Aspekte hinzufügt, wie z. B. syntaktische oder semantische Korrektheit oder stilistische Fragen. Grappa soll diese **Aufgabenstruktur** kennen und persistent speichern. Um die Bewertung durchführen zu können, benötigt der Grader meist neben der Einreichung eine **Aufgabenbeschreibung und weitere Konfigurationsdateien**. Diese soll Grappa ebenfalls persistent zu den hinterlegten Aufgaben vorhalten und dem Grader passend zur Verfügung stellen. Dazu wird eine Konfigurationsschnittstelle benötigt.

Bewertungskommentare können entweder allgemein sein oder spezifisch unterschieden werden zwischen Feedback für die Studierenden und Dozenten. Sie können sich entweder auf die gesamte Aufgabe beziehen oder auf eine Teilaufgabe bzw. einen Aspekt. Weiterhin muss Grappa zwischen den vom LMS und vom Grader unterstützen Formaten wie z. B. PDF, XML, HTML und Text vermitteln. Wenn möglich sollte Grappa auch zwischen den Formaten konvertieren.

Die **Bewertungsskala** kann sich zwischen LMS und Grader unterscheiden. Ein Grader könnte z. B. alle Aufgaben auf einer Skala zwischen 0 und 100 Punkten bewerten, wohingegen das LMS erwartet, dass zu der Aufgabe passende Punkte oder Noten vergeben werden. Hier muss Grappa ggf. mit Unterstützung des LMS eine entsprechende Umrechnung vornehmen können.

Je nach Art der Aufgabe kann ein Grader merklich Zeit für die Bewertung benötigen. Damit trotzdem eine zügige Verarbeitung bei **paralleler Nutzung** (z. B. im Rahmen einer Übungsstunde) mo¨glich ist, sollte Grappa in der Lage sein, Einreichungen parallel vom LMS anzunehmen und diese parallel an den Grader zur Bewertung schicken. Hierbei ist natürlich zu beachten, ob der Grader die parallele Nutzung unterstützt. Weiterhin muss das LMS einen Grader **asynchron** aufrufen können sowie laufende Bewertungsvorgänge abfragen und abbrechen können.

Die Kopplung eines weiteren LMS-Typs mit Grappa muss einfach und ohne großen Aufwand möglich sein. Dazu ist es erforderlich, **Clientbibliotheken** anzubieten, welche die technischen Details der Ansteuerung von Grappa weitgehend kapseln. Anfänglich wird eine solche Bibliothek für php angeboten.

Manche Grader wie z. B. aSQLg erlauben die **Parametrierung von Aufgaben**. Dies bedeutet, dass aus einer Aufgabenvorlage (Template) per Zufall eine konkrete Ausprägung einer Aufgabe erstellt wird. Dies dient dem Generieren von zusätzlichen Übungsaufga-

ben aus wenigen Vorlagen und dem Schutz vor Abschreiben. Aus studentischer Sicht ist es wünschenswert, im LMS die erneute Generierung einer Aufgabe anzustoßen, um zusätzliche Varianten zum Üben zu bekommen. Für Grappa bedeutet dies, dass bei der Bewertung die zu bewertende Aufgabenvariante bekannt sein muss. Weiterhin wird eine Funktion benötigt, mit der das LMS vom Grader eine neue Variante einer Aufgabe anfordern kann. Diese muss dann auch einen an die Variante angepassten Aufgabentext enthalten, der vom LMS angezeigt werden kann. Auch hier ergibt sich somit das oben genannte Problem der Formate.

4 Fachdatenmodell und Kernfunktionen (M)

In diesem Abschnitt beschreiben wir das Fachdatenmodell von Grappa. Zur Erläuterung und Motivation des Fachdatenmodells integrieren wir an geeigneten Stellen eine Beschreibung der von Grappa implementierten Funktionen des Aufgabenbereichs (M) (vgl. Abschnitt 2). Querbezüge zu den Anforderungen (Abschnitt 3) werden **fett** gesetzt. Denkbare Realisierungsalternativen werden als Fußnoten gesetzt.

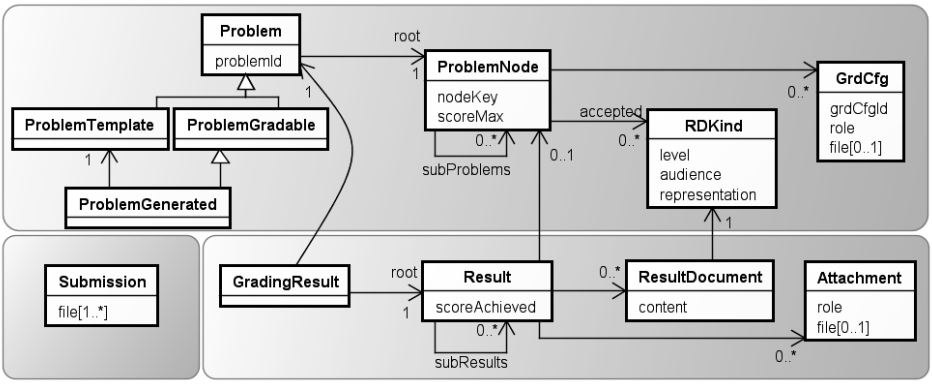


Abb. 3: Ausschnitt des Grappa-Fachdatenmodells mit den drei Bereichen **Aufgabenstruktur und –konfiguration** (*Problem*, *GrdCfg*), **Einreichung** (*Submission*) und **Bewertungsergebnis** (*GradingResult*). Attribute sind aus Platzgründen nur vereinzelt dargestellt.

Die **Aufgabenstruktur** wird im Fachdatenmodell (vgl. Abb. 3) von einem *Problem* repräsentiert. Die Lehrkraft erzeugt mit LMS-Unterstützung ein *Problem*-Objekt zusammen mit hierarchisch untergeordneten *ProblemNode*s für eventuelle Teilaufgaben. Ein Baum von *ProblemNode*s ist vorteilhaft, um Teilaufgaben bzw. Teilaspekte der Aufgabenstruktur flexibel und auf mehreren Stufen bündeln zu können. Beispielsweise lässt sich eine Java-Programmieraufgabe (= Wurzel-*ProblemNode*) abbilden, die mehrere zu implementierende Klassen in je einer Teilaufgabe (= *ProblemNode* der zweiten Ebene) beschreibt. Für jede Teilaufgabe wiederum können einzelne Bewertungsaspekte (syntaktische Korrektheit, semantische Korrektheit, Wartbarkeit, ...) in *ProblemNode*s der drit-

ten Ebene dargestellt werden.[8]

Die Klassenhierarchie unterhalb von *Problem* modelliert alle Arten von Aufgaben. Ein *Problem* kann dabei entweder eine bewertbare Aufgabe sein (*ProblemGradable*) oder ein Aufgabentemplate (*ProblemTemplate*). Eine aus einem Template generierte Aufgabe (*ProblemGenerated*) ist ein Spezialfall einer bewertbaren Aufgabe. Ein generiertes *Problem* speichert einen Verweis auf das Template, aus dem es generiert wurde, um ein späteres erneutes Generieren zu erlauben. Ein Template selbst kann nicht bewertet werden.

Die **Bewertungsskala** in Gestalt zu vergebender Maximalpunkte sowie eventuelle Teilaufgabenschlüssel werden von der Lehrkraft vorgegeben. Die *ProblemNode*-Hierarchie nutzt Grappa, um ein initiales Bewertungsergebnis (*GradingResult*) strukturgleich mit Teilergebnissen (*Result*) vorzubereiten. Der Grader kann die vorgegebene Ergebnisstruktur durch *Result*-Objekte auf weiteren, unteren Hierarchie-Ebenen ohne zugehöriges *ProblemNode*-Objekt ergänzen. Wenn der Grader die erreichten Punkte (*scoreAchieved*) an den Blättern des *Result*-Baums vermerkt hat, berechnet Grappa für innere Knoten und damit auch die Gesamteinreichung die aggregierten Punktzahlen. Sollte das LMS eine nicht punktebasierte Bewertungsskala nutzen, wäre eine Umrechnung im LMS-Plugin umsetzbar (vgl. auch Diskussion in Abschnitt 7).

Das LMS oder die Lehrkraft legen fest, welcher „Art“ die zurück erwarteten **Bewertungskommentare** sein sollen. Auf dem Hinweg LMS→Grappa→Grader werden Wünsche in Form von *RDKind*-Objekten (RD=result document) transportiert. Auf dem Rückweg liefert Grappa möglichst zu jedemWunsch ein passendes *ResultDocument*. Die Art eines Bewertungskommentars kann in den Dimensionen Detailgrad (*level*), Zielgruppe (*audience*) und Format (*representation*) unterschieden werden.

Die vom Grader gelieferten Kommentarinhalte werden von Grappa durch *ResultDocument*-Objekte genau dem zugehörigen Bewertungsergebnis (*Result*) zugeordnet. Dies können unterschiedliche Formate wie Texte oder Bilder sein. Das LMS erhält so die Möglichkeit, Kommentar und Punktzahl im Zusammenhang darzustellen, etwa in Gestalt eines für den einreichenden Studenten benutzerfreundlich navigierbaren Baumes mit zu jeder Teilaufgabe aufklappbaren Detailkommentaren.[9]

Zur Motivation dieses Datenmodells skizzieren wir beispielhaft den folgenden Wunsch einer Lehrkraft nach drei verschiedenen Bewertungskommentaren:

[8] Realisierungsalternative: Eine flache Listenstruktur von Teilaufgaben bzw. Bewertungsschritten wie in [SSM+14] wurde verworfen, weil sie im Gegensatz zu einem Baum schlecht zur gebündelten Konfiguration mehrstufig gruppierter Aufgabenaspekte geeignet ist.

[9] Realisierungsalternative: Statt eines gemäß der Aufgabenstruktur aufgebauten Ergebnisses hätte man das Ergebnis davon unabhängig strukturieren können - z. B. als ein Gesamtdokument mit dem vollständigen Bewertungsergebnis. Nachteil: das LMS könnte Teilergebnisse nicht ohne weiteres im Kontext der jeweiligen Teilaufgabe darstellen. Mit einem nach Bewertungsaspekt, Zielgruppe und Detailgrad strukturierten Ergebnis kann das LMS dieses benutzerspezifisch gefiltert anzeigen und so den Benutzungskomfort erhöhen.

- ein normaler Bewertungskommentar im HTML-Format mit Hinweisen zur Verbesserung von Programmierfehlern. Diesen Kommentar präsentiert das LMS dem einreichenden Studenten unmittelbar nach dem Ende des Bewertungsvorgangs.
- ein minimaler Bewertungskommentar im Format „plain text“ (bspw. mit dem Inhalt „Fehlerhafte Einreichung“ oder „keine Einreichung“), den das LMS der Lehrkraft in einer Gesamtübersicht aller einreichenden Studierenden nach Ende der Abgabefrist automatisch per E-Mail zusendet,
- zusätzlich einen ausführlichen Kommentar im HTML-Format mit Meldungen, die den Bewertungsablauf beschreiben, sowie ggf. mit in den Kommentar eingefügten Musterlösungen. Diesen Kommentar zeigt das LMS nur der dazu autorisierten Lehrkraft an, die so die vorgenommene Bewertung nachvollziehen kann.

Als „extension point“ für Bewertungsergebnisse lassen sich die an ein *Result*-Objekt geknüpften *Attachment*s verstehen. Diese graderspezifischen Dateiobjekte enthalten bspw. detaillierte Graderprotokolle, die vom LMS z. B. für spätere Statistikzwecke aufbewahrt werden können.

Eine **studentische Einreichung** ist durch *Submission* repräsentiert. Grappa erwartet hier Dateien, wobei das LMS Texteingabefelder vorsehen kann, die vom LMS-Plugin in Dateien zu konvertieren sind.

Konfigurationsdateien dienen der Konfiguration eines Graders. Darunter verstehen wir Dinge wie eine jar-Bibliothek, eine JUnit-Testklasse, eine aSQLg-Musterlösung, eine Textdatei mit einer Datenbank-URL, usf. *GrdCfg* speichert eine Konfigurationsdatei mit einer im Graderkontext verwertbaren Bedeutung (*role*). Grappa kann das LMS mit verfügbaren roles versorgen, um die Auswahl für die Lehrkraft, die die Konfigurationsdateien ins LMS einstellen muss, benutzerfreundlich zu gestalten. *GrdCfg*-Objekte können sich auf einzelne Teilaufgaben / -aspekte beziehen oder auf die Gesamtaufgabe. In einer SQLAufgabe mit mehreren geforderten SQL-Anweisungen bezieht sich das *GrdCfg*-Objekt zur Konfiguration der Datenbankverbindung in der Regel auf den Wurzel-*ProblemNode* während Musterlösungen für jede einzelne SQL-Anweisung als *GrdCfg*-Objekte jeweils mit Bezug auf einen *ProblemNode* der zweiten Baumebene realisiert werden.[10]

Der Grappa-Kern (M) interpretiert *GrdCfg*-Objekte nicht. Denkbar ist eine Interpretation im (B)ackend.

Grappa persistiert *Problem*- und *GrdCfg*-Objekte, um nachfolgende studentische **Einreichungen** mit weniger Kommunikationsballast zwischen LMS und Grappa abwickeln zu können. Ein weiterer Gedanke bei der Persistierung war die kursübergreifende Verwen-

[10] Realisierungsalternative: die Idee, Konfigurationsparameter nicht einfach durch eine Dateisammlung, sondern stärker formalisiert darzustellen, wurde zugunsten der Anwendbarkeit auf beliebige Grader fallengelassen. Zudem wurde der *role*-Wertebereich anders als in [SSM+14] nicht eingeschränkt, weil eine graderübergreifende Verwendung der *role* nicht zum Anforderungskatalog gehört.

dung von **Aufgabenstrukturen** und **Konfigurationsdateien**.[11]

Um einen **asynchronen Bewertungslauf** zu realisieren, implementiert der Grappa-Kern (M) einen Pool wartender bzw. sich in Bewertung befindlicher Anfragen. Der Pool verfügt über Funktionen zur Statusabfrage und zum Abbrechen eines Vorgangs.

5 Schnittstellen

Zunächst beschreiben wir von Grappa implementierte Funktionen des Aufgabenbereichs (F). Querbezüge zu Anforderungen (Abschnitt 3) werden **fett** gesetzt. Dem LMS bietet Grappa drei Gruppen von Operationen als REST-Schnittstellen an. Zum einen (vgl. Abb. 4) bietet *Setup* CRUD-Operationen für *Problem*- und *GrdCfg*-Objekte an. Die in Abb. 3 ersichtlichen Attribute *problemId* und *grdCfgId* ermöglichen dem LMS die nachträgliche Identifizierung dieser Objekte. Es ist geplant, dass Grappa auf Wunsch auf die Vorab-Persistierung von *Problem*- und *GrdCfg*-Objekten verzichtet.

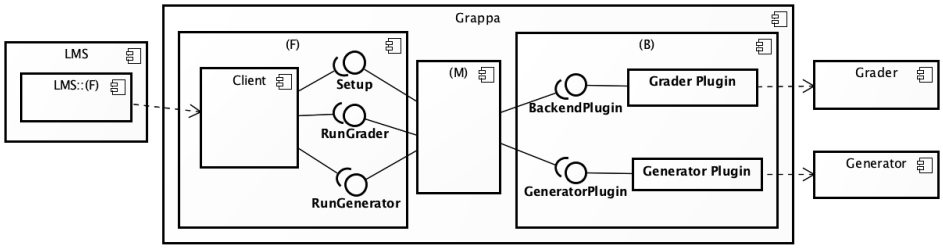


Abb. 4: Schnittstellenkonzept

Die *RunGrader*-Schnittstelle behandelt **Einreichungen**, welche sowohl synchron als auch asynchron vom LMS an Grappa übermittelt werden können. Neben dem *Submission*-Objekt übermittelt das LMS die für die Bewertung heranzuziehende *problemId*. Zu asynchronen Einreichungen liefert Grappa unmittelbar als Antwort die erwartete Restdauer des Bewertungsvorgangs, so dass das LMS seine nachfolgenden Ergebnisanfragen (Polling) angemessen takten kann. Das Polling kann für das LMS völlig transparent in einem von Grappa bereit gestellten und an die LMS-Erfordernisse anpassbaren Client erfolgen, der bei Vorliegen des Ergebnisses dem LMS eine Nachricht zukommen lässt.

Über die *RunGenerator*-Schnittstelle fordert das LMS eine neue Variante zu einem Aufgaben-Template an. Die Variante wird über das *GeneratorPlugin* erzeugt, intern gespeichert und zurückgegeben. Eine erzeugte Aufgaben-Variante kann später neu generiert

[11] Realisierungsalternative: Wenn eine kursübergreifende Verwendung von Aufgaben nicht angestrebt wird und wenn die Größe der *Problem*- und *GrdCfg*-Objekte gering ist, kann die Persistierung entfallen. Das LMS muss dann bei jeder studentischen Einreichung neben der *Submission* auch die zugehörigen *Problem*- und *GrdCfg*-Objekte übermitteln. Dies hat den Vorteil, dass doppelte Datenhaltung in LMS und Grappa und die damit einhergehende Gefahr von Inkonsistenzen vermieden würde.

und durch die neue Variante ersetzt werden, falls der Student die Aufgabe zum Üben erneut bearbeiten will. Dabei kann auch nur ein Teil der Aufgabe neu generiert werden.

Nun wenden wir uns den Funktionen des Aufgabenbereichs (B) zu. Grappa integriert Grader über eine *BackendPlugin*-Schnittstelle. Ein konkretes *GraderPlugin* übernimmt die technische und fachliche Anbindung des Graders an Grappa. Abhängig von den graderspezifischen Erfordernissen nutzt das *GraderPlugin* einmalig in Grappa realisierte Funktionsbibliotheken, um z. B. eine Umrechnung der Grader-**Bewertungsskala** in die Grappa-Punkteskala sowie ggf. Umformatierungen von **Bewertungskommentaren** vorzunehmen. Weiterhin informiert das *GraderPlugin* den Grappa-Kern über die Möglichkeit der **parallelen Nutzung** des Graders in Form einer Maximalanzahl zeitgleich durchgeführter Bewertungen. Um die Erzeugung von konkreten Aufgaben aus Aufgaben-Templates zu unterstützen, bietet Grappa die *GeneratorPlugin*-Schnittstelle als Verbindung zwischen Grappa und den unterschiedlichen Aufgaben-Generatoren an.

6 Verwandte Arbeiten

Bestehende Ansätze zur Integration von LMS und Gradern sind häufig entweder konzipiert wie etwa Web-CAT [Edw03] oder wie moodle VPL [RRH12]. Die beiden vorgenannten Systeme sollen hier als Vertreter jeweils einer ganzen Reihe ähnlich konzipierter Systeme exemplarisch heraus gegriffen werden.

Web-CAT entstand um eine Bewertungsidee herum (Heranziehung der von Studenten geschriebenen Tests bei der Bewertung) und bietet diese Bewertungsidee für verschiedene Programmiersprachen an. Um einen bequemen Zugang zu ermöglichen, steuert Web-CAT die Bewertung mit einem eigens dafür konzipierten LMS mit per Webbrowser bedienbaren Kursverwaltungsfunktionen. Ohne weiteren Aufwand ist es nicht möglich, den in Web-CAT genutzten Autobewerter in fremden LMS zu nutzen.

Das Plugin moodle VPL ist (wie der Name schon sagt) eng in das weit verbreitete LMS moodle integriert und erweitert dieses um die Möglichkeit, Programmieraufgaben in diversen Programmiersprachen zu bewerten. Die enge Integration bietet Vorteile bei der Nutzung, verwehrt jedoch die Nutzung der Autobewerter mit einem anderen LMS. Grappa verfolgt den Ansatz, den handelnden Personen die Wahl des Systems zur Kursverwaltung nicht vorzugeben, wenn sie automatisierte Programmbewertung einsetzen wollen.

Eine mit dem in Abschnitt 2 beschriebenen Konzept vergleichbare Architektur klingt in [PJR12] an, wobei dort eher didaktische und organisatorische Aspekte beleuchtet werden, die auf Erfahrungen mit einer prototypischen Implementierung basieren.

In [SBG09] wird eine modulare Softwarearchitektur beschrieben, die verschiedene marking components an verschiedene Frontends (webbasiert oder rich clients) anbindet. Zur automatischen Bewertung einer einzigen Einreichung führt das System JACK, welches diese Architektur realisiert, ggf. mehrere marking components aus, um verschiedene As-

pekte (z. B. Syntax, Stil) zu überprüfen. Trotz der Ähnlichkeiten zu Grappa hat JACK eine gänzlich andere Zielsetzung, nämlich die Abwicklung von ganzen Prüfungen inkl. der Verwaltung von Prüfungskandidaten, -inhalten und -terminen, inkl. der Speicherung von Einreichungen und inkl. einer manuellen Bearbeitung von Autobewerter-Ergebnissen. Dieser Zielsetzung entspricht auch die Unterstützung weiterer Aufgabentypen wie Essays und Auswahlantwortaufgaben. Grappa zielt hingegen ausschließlich auf den Bewertungsprozess und überlässt die Prüfungsverwaltung dem LMS. Es gibt derzeit Bestrebungen, die JACK-Teilkomponente, die den automatischen Bewertungsprozess steuert, durch ein BackendPlugin an Grappa anzuschließen, um einen moodle-Zugang zu ermöglichen.

In [AKR11] werden Autobewerter als Backends über einen zentralen ECSpooler mit LMSen (den Frontends) in einer serviceorientierten Architektur (SOA) verbunden. Anders als bei Grappa sind hier Backends lose über Standard-Internetprotokolle an den ECSpooler gekoppelt. Legacy-Autobewerter müssen dazu separat mit einer Webservice-Schnittstelle ausgestattet werden. Grappas BackendPlugin-Konzept hingegen kann Aufwände zur Einbindung von Autobewertern reduzieren, die nicht über eine Webschnittstelle verfügen. Der ECSpooler erwartet Frontends, die ihre Benutzerschnittstelle zur Aufgabendefinition dynamisch an den jeweils eingesetzten Autobewerter anpassen. Mit Grappa fokussieren wir für das LMS auf eine einheitliche und damit aufwandsärmer erstellbare Standard-Benutzerschnittstelle, die das im vorliegenden Beitrag beschriebene Fachdatenmodell abbildet.

Dreh- und Angelpunkt zur Umsetzung der oben beschriebenen Architekturidee ist das konzeptionelle Modell der Daten, die eine Programmieraufgabe, studentische Einreichungen und das automatisch generierte Feedback beschreiben. In [PJR13] wird ein konzeptionelles Modell einer Programmieraufgabe vorgestellt, welches Erstellung, Suche, Austausch, Evaluierung und Verbesserung der Programmieraufgabe unterstützen soll. Vergleichbar sind Bestrebungen im eCULT-Projekt[12], ein universelles „Austauschformat“ für den Transfer von Programmieraufgaben zwischen den beteiligten LMS und Gradern zu entwickeln [SSM+14]. Da das konzeptionelle Modell bzw. das Austauschformat in seiner Zielsetzung von Programmiersprache und LMS unabhängig sind, können diese als Beschreibung aller Daten aufgefasst werden, die ein beliebiges LMS mit einem beliebigen Grader für die automatisierte Bewertung austauschen muss.

Das in Grappa verwendete Fachdatenmodell (vgl. Abschnitt 4) besitzt Ähnlichkeiten und Unterschiede zu dem oben zitierten konzeptionellen Modell [PJR13], wobei unser Modell an einigen Stellen weiter abstrahiert (bspw. ist eine Referenz auf eine Quellcodezeile im Feedback für Modellierungssprachen wie UML nicht sinnvoll), an einigen Stellen weniger Ausdrucksmöglichkeiten als jenes besitzt (bspw. ist das menschliche Feedback einer Lehrkraft in unserem Modell nicht abgebildet), und wiederum an einigen Stellen detaillierter auf technische Besonderheiten und Zusatzfunktionen des Graders eingeht (bspw. kann in Grappa das Feedback um Daten zu Format, Detailgrad und Zielgruppe

[12] http://www.ecult-niedersachsen.de/

ergänzt werden).

Auch im Vergleich zum oben zitierten Austauschformat [SSM+14] besitzt das Grappa Fachdatenmodell Besonderheiten. So erlaubt Grappa über das Austauschformat hinaus die hierarchische Strukturierung einer Aufgabe in Teilaspekte. Zudem wird im Austauschformat kein Standard für die quantitative Bewertung einer Einreichung in Form einer Punktzahl o. ä. vorgeschlagen. Das Grappa Fachdatenmodell schlägt die Vorgabe einer maximal erreichbaren Punktzahl für jeden Teilaspekt einer Aufgabe vor. Durch die Grappa-Plugin- Architektur kann diese Vorgabe an graderspezifische Erfordernisse angepasst werden.

7 Zusammenfassung und Ausblick

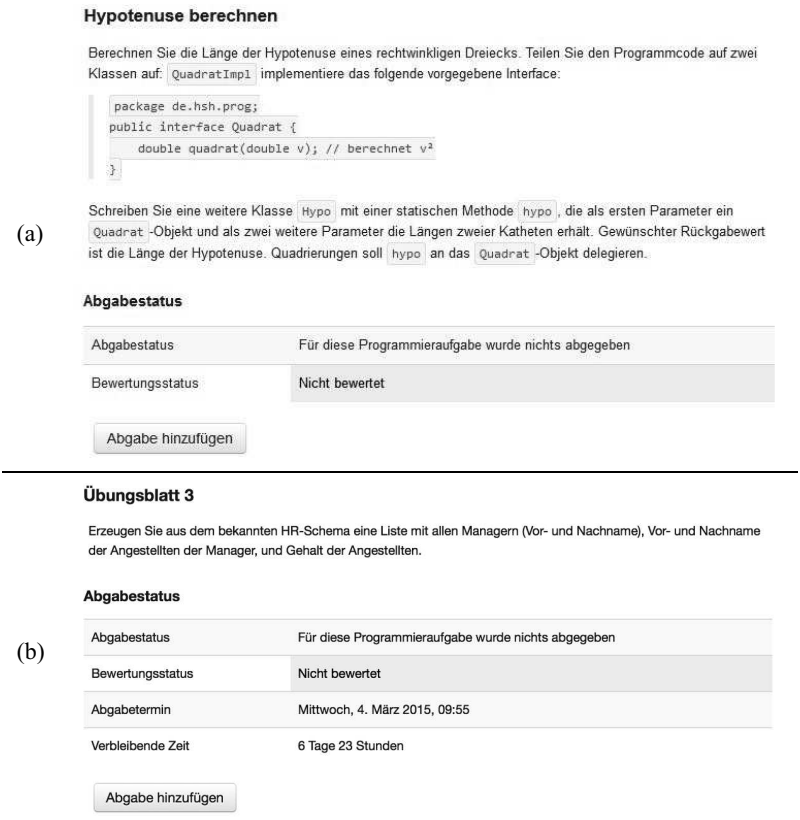


Abb. 5: Homogene Bedienung zweier Grader in einer an Grappa angeschlossenen moodle-Installation. Aufgabenstellung mit der Möglichkeit der Lösungseinreichung zu einer Java-Aufgabe (a) bzw. einer SQL-Aufgabe (b).

Bisher mussten zur Anbindung von Autobewertern an LMSe jeweils spezielle Adapter entwickelt werden. Durch die Verwendung der in diesem Beitrag vorgestellten Middleware Grappa lässt sich dieser Aufwand auf ein Minimum reduzieren. Grappa bietet eine wiederverwendbare Implementierung der typischen Aufgabenstellungen bei der Anbindung. Dazu gehören u. a. Mechanismen zur Konfiguration der Aufgaben, zum Abrufen der Bewertungen, zur Konvertierung von Punkten und Kommentaren, sowie zum parallelen Aufruf der Bewertung. Grappa versetzt ein LMS wie moodle in die Lage, Studierenden eine homogene Bedienung verschiedener Grader anzubieten (vgl. Abb. 5).

Grappa definiert ein allgemeines Modell für Aufgaben, welches unabhängig vom Typ der Aufgabe ist. Eine Abstimmung mit dem in [SSM+14] entwickelten „Austauschformat" von Aufgaben wird derzeit vorgenommen. Insbesondere die Frage, wie eine Aufgabe in Teilaspekte unterteilt werden kann und wie die einzelnen Aspekte bei der Bewertung gewichtet werden, ist in Grappa derzeit auf einfache Weise gelöst (Baumstruktur mit Punktgewichten an Baumknoten), im Austauschformat wird diese Frage derzeit weitgehend ausgeklammert. Eine Erweiterung des Grappa-Modells um komplexere, formelbasierte Bewertungsgewichtungen ist denkbar. Beispielhaft sei hier der Grader JACK[13] genannt, der es erlaubt, sog. Evaluierungsregeln zu formulieren. Ebenfalls ist es denkbar, LMS-spezifische Bewertungsskalen (auch nicht-numerische) in Grappa abzubilden. Obwohl denkbar, planen wir jedoch zunächst bei der einfachen Punktebewertung zu bleiben. LMSe wie moodle besitzen nämlich bereits mächtige Funktionen zur Umrechnung von Punkten in andere Bewertungsskalen. Und auch Grader wie JACK können problemlos in Grappa integriert werden, indem die Umrechnungen zwischen dem Grappa-Bewertungsansatz und dem JACK-Bewertungsansatz im JACK-Backend-Plugin vorgenommen wird.

Der Grader benötigt zum Bewerten von Aufgaben i. d. R. eine passende Umgebung. So muss z. B. für SQL-Aufgaben ggf. ein passendes Datenbankschema eingerichtet und befüllt sein. Für Programmieraufgaben sind ggf. bestimmte Bibliotheken Voraussetzung. Ein Import entsprechender Aufgaben setzt derzeit die manuelle Konfiguration der Umgebung des Graders voraus. Um dieses zu vereinfachen, arbeiten wir an einem Konzept, mit welchem benötigte Ressourcen wie Bibliotheken oder Schemata aus einem Repository nachgeladen und installiert werden können.

Grappa ist bereits erfolgreich an der Hochschule Hannover im Einsatz. Dies wäre ohne die tatkräftige Unterstützung aller Grappa-Teammitglieder nicht möglich gewesen, für die wir uns herzlich bedanken. In [SBG+14] beschreiben wir eine Evaluation des Gesamtsystems, welche zeigt, dass trotz des hohen Abstraktionsgrades der Schnittstellen eine benutzerfreundliche Integration in ein LMS gelingt.

[13] http://www.s3.uni-duisburg-essen.de/forschung/e-learning-and-e-assessment/jack/

Literaturverzeichnis

[AKR11] Amelung, M.; Krieger, K.; Rosner; D.: E-Assessment as a Service. Learning Technologies, IEEE Transactions on, 4/11, S. 162–174, 2011.

[Edw03] Edwards, S.: Using test-driven development in the classroom: Providing students with automatic, concrete feedback on performance. In: Proceedings of the International Conference on Education and Information Systems: Technologies and Applications EISTA, 3, 2003.

[Gar13] Garmann, R.: Sicherheitsimplikationen beim Einsatz von Test Doubles zur automatisierten Bewertung studentischer Java-Programme mit Graja und mockito. In: Workshop „Automatische Bewertung von Programmieraufgaben“, 28.10.2013, Hannover, 2013.

[KTH13] Kleiner, C.; Tebbe, C.; Heine, F.: Automated Grading and Tutoring of SQL Statements to Improve Student Learning. In Proceedings of the 13th Koli Calling International Conference on Computing Education Research, S. 161–168, 2013.

[PJR12] Priss, U.; Jensen, N.; Rod, O.: Software for E-Assessment of Programming Exercises. In: GI-Jahrestagung, S. 1786–1791, 2012.

[PJR13] Priss, U.; Jensen, N.; Rod, O.: Using Conceptual Structures in the Design of Computer-Based Assessment Software. In: ICCS, S. 121–134, 2013.

[RRH12] Rodríguez-del Pino, J.; Rubio-Royo, E.; Hernández-Figueroa, Z.: A Virtual Programming Lab for Moodle with automatic assessment and anti-plagiarism features. In: Proceedings of the 2012 International Conference on e-Learning, e-Business, Enterprise Information Systems, & e-Government, 2012.

[SBG09] Striewe, M.; Balz, M.; Goedicke, M.: A Flexible and Modular Software Architecture for Computer Aided Assessments and Automated Marking. In CSEDU (2), S. 54–61, 2009.

[SBG+14] Stöcker, A.; Becker, S.; Garmann, R.; Heine, F.; Kleiner, C.; Werner, P.; Grzanna, S.; Bott, O.: Die Evaluation generischer Einbettung automatisierter Programmbewertung am Beispiel von Moodle und aSQLg. In: DeLFI, S. 301–304, 2014.

[SSM+14] Strickroth, S.; Striewe, M.; Müller, O.; Priss, U.; Becker, S.; Bott, O.; Pinkwart, N.: Wiederverwendbarkeit von Programmieraufgaben durch Interoperabilität von Programmierlernsystemen. In: DeLFI, S. 97–108, 2014.