Programmierausbildung Online

H. Eichelberger, G. Fischer, F. Grupp, J. Wolff v. Gudenberg
wolff@informatik.uni-wuerzburg.de

1 Einleitung

Programmieren ist ein kreativer Vorgang. Mehr als auf die bloße Beherrschung der Syntax der Sprache kommt es auf geschickte Auswahl der Datentypen, gute Strukturierung, die richtige Zerlegung in kleine Einheiten und die Fähigkeit existierenden Quellcode einzubinden und das eigene Programm so zu formulieren, dass Teile davon wieder verwendet oder weiter verwertet werden können, an. Es ist wichtig die der Programmiersprache zu Grunde liegenden Konzepte [2, 4] zu verstehen und umsetzen zu können.

Programmieren kann als ein Handwerk aufgefasst werden, in dem feste Regeln und Konventionen gelten. In letzter Zeit hat sich wie im Handwerk ein Trend ausgebildet, weg von der handwerklichen Erstellung eines Programms von Anfang an, hin zu einer Zusammenstellung und Anpassung vorliegender Halbfabrikate. Wir glauben jedoch, dass die Fähigkeit die Komponenten von Grund auf zu verstehen und auch schreiben zu können unbedingt von einem Informatiker beherrscht werden muss.

In diesem Artikel wollen wir Kriterien aufstellen, wie eine Programmierausbildung online über das Internet auszusehen hat, um im Selbststudium Erfolg versprechend eingesetzt werden zu können. Dabei ist die automatische Beurteilung eingereichter Programmieraufgaben wie in [8] ein wesentlicher Punkt, der unseren Ansatz (siehe auch [7, 5]) von den meisten anderen (z.B. [6]) unterscheidet.

Wir erläutern dann die Umsetzung dieser Kriterien am Beispiel des Java Online Praktikums (JOP) [5] der vhb[1].

2 Kriterien für eine Online Programmierausbildung

In einer Online Selbstlernumgebung kommt es auf neben der attraktiven Präsentation des Lerntextes [9] auf ein hohes Maß an Interaktivität bei den Beispielen und besonders auf angemessene Behandlung der Programmieraufgaben an. Die Benutzerführung, Anpassbarkeit und die Leistungskontrolle spielen ebenfalls eine Rolle. In [10] werden 8 Imperative für virtuelles Lernen aufgestellt, die wir im Wesentlichen unterstützen und umsetzen.

[1] http://www.vhb.org

Zur Umsetzung dieser Kriterien und auf Grund unserer Erfahrungen im Gebrauch und Einsatz von Online Tutorien stellen wir folgende konkrete Forderungen auf:

- Der Text sollte aufbereitet und durch Bilder und Grafiken aufgelockert sein.
- Der Text sollte knapp sein.
- Der Text sollte verlinkt sein.
- Animationen erläutern schwierige Algorithmen und Strukturen.
- Direkte Anbindung der Standard-Dokumentation und eines Glossars.
- Viele interaktiv ausführbare Beispiele dienen zur Erläuterung.
- Die Beispiele lassen sich verändern und erneut ausführen.
- Modifikation und Ausführung der Beispiele geschieht innerhalb des Systems, d.h. ohne dem Benutzer zusätzliche Schritte abzufordern.
- Es werden Aufgaben gestellt, und die eingereichten Lösungen werden direkt ausgewertet und beurteilt.
- Diese Auswertung beschränkt sich nicht nur auf Multiple-Choice-Tests und Lückentexte oder Freitexteingaben, sondern wird auch für Programmieraufgaben durchgeführt.
- Umfangreiche Tests überprüfen die Funktionalität.
- Detaillierte Meldungen helfen dem Lernenden seine Lösung zu verbessern.
- Eine Folge von Hinweisen (Tipps) unterstützt den Lernenden schrittweise auf dem Weg zur Lösung.
- Auch die Form der eingereichten Quelltexte wird beurteilt.
- Der Benutzer kann verschiedene (vorgefertigte) Lernpfade durch den Inhalt wählen.

Diese Kriterien wurden in JOP im Wesentlichen umgesetzt. Der Lernende hat in diesem Tutorial jederzeit die Möglichkeit den vorgeschlagenen Lernpfad zu verlassen und nach eigenen Vorstellungen weiter zu machen. Programmieraufgaben werden vom Praktomat (siehe 3.5) überprüft.

Die eigentliche Leistungskontrolle des Praktikums erfolgt in JOP durch das Lösen von relativ umfangreichen Aufgaben im Praktomat.

3 Java Online Praktikum (JOP)

3.1 Überblick

JOP besteht aus zwei Teilen:

Im Tutorial wird das objektorientierte Programmieren mit Java vermittelt, wobei die Studenten die Beispiele des Tutorials direkt oder nach Veränderung ausführen und Kontrollaufgaben bearbeiten können.

Die eingereichten Java Programme werden durch moderne Verfahren der Programmanalyse (≫elektronische Tutoren≪) automatisch nach Funktionalität und Lesbarkeit bewertet. Studenten erhalten so direkte Rückmeldung über ihren Lern- und Programmiererfolg.

Im anschließenden Praktikum werden größere Java-Programme erstellt; erneut sichern die elektronischen Tutoren die Eingangsqualität der Programme. Die abschließende Bewertung erfolgt durch Mitarbeiter, unterstützt durch elektronische Tutoren.

3.2 Lerntexte

Die einzelnen Lerneinheiten werden als XML Texte dynamisch dargestellt. Dabei sind verschiedene Sichten auf eine Einheit möglich, z.B. gibt es eine Überblicksicht und eine Referenz mit Syntaxdiagrammen und Semantikregeln. Es gibt Lernpfade für Einsteiger (die noch keine Programmierkenntnisse haben), Umsteiger (die nur die objektorientierte Programmierung erlernen wollen) und Nebenfächler (die zuweilen eine etwas vereinfachte Sicht bevorzugen).

3.3 Beispiele

Beispiele enthalten eigentlich immer Java Programme von denen nur die relevanten Ausschnitte aufbereitet gezeigt werden. Für alle Beispiele liegt aber auch immer der gesamte Code für ein ausführbares Programm vor. Dieser kann angezeigt oder auch im Browser oder einem Editor verändert werden, dann direkt übersetzt und ausgeführt werden.

Dabei wird das (möglicherweise abgeänderte) Programm zunächst an den Server geschickt, um es dort zu kompilieren und mit zusätzlich benötigten Hilfsklassen, Ressourcen und Bibliotheken zu versehen. Das fertig übersetzte und gepackte Beispiel-Programm wird dann an den Client geschickt, der es mit Hilfe der JAVA WEB START-Applikation[2] JSEX in einer sicheren Umgebung ausgeführt.

Die Ausführungsumgebung JSEX simuliert eine Kommandozeile zur Interaktion mit dem Programm durch Umleiten der Ein- und Ausgabeströme und ermöglicht mit Hilfe des

[2] http://java.sun.com/products/javawebstart/download.html

Pakets `simple`[3] auch gesicherte Lese- und Schreibzugriffe auf Dateien. Auch grafische, also z.B. AWT oder Swing Applikationen können ausgeführt werden.

3.4 Aufgaben

Es gibt 3 Typen von Aufgaben

- Multiple-Choice: Eine oder mehrere Antworten können richtig sein.
- Freitext: Der eingegebene Text, oft eine Zahl oder ein eng eingegrenzter Ausdruck wird anhand eines regulären Ausdrucks auf Übereinstimmung mit der Lösung überprüft.
- Programmieraufgaben: Diese werden intern an den Praktomat weiter gereicht, der aufgabenspezifisch konfiguriert ist.

Die Lösungen werden in allen Fällen sofort ausgewertet, und die Ergebnisse mit entsprechenden fehlerspezifischen Hinweisen an den Benutzer gemeldet. Es können zusätzlich Hinweise (Tipps) spezifiziert werden, die dem Benutzer bei entsprechenden Fehlversuchen angeboten werden. Auch die Musterlösung wird bei einer korrekten Antwort oder bei einer bestimmten Anzahl von Fehlversuchen zur weiteren Erläuterung bereit gestellt.

3.5 Praktomat

Für JOP wurde der ursprüngliche, in Passau entwickelte Praktomat[11] um Java-spezifische elektronische Tutoren erweitert, lediglich die Prüfung auf Übersetzbarkeit und die Blackbox-Tests mit DejaGnu konnten fast unverändert übernommen werden. Im neuen Praktomat[7] sind zu jeder Aufgabe eine Reihe von Prüfungen definiert, die bei der Auswertung einer Lösung ausgeführt werden.

- **Übersetzbarkeit:** Das Programm muss übersetzt werden können. Dazu wird einfach der Compiler aufgerufen, und eventuelle Fehlermeldungen ausgewertet.
- **Kodierungskonventionen:** Der Kodierungsstil entspricht bzgl. Einrückung und Namensgebung den geforderten Standards.

  Diese Standards lassen sich in einer Konfigurationsdatei angeben. Das Programm wird mit einem speziellen Parser[3] geparst, der die Positionen der Konstrukte im Quelltext feststellt und auch die Kommentare berücksichtigt. Die Aufgabe dieses elektronischen Tutors kann teilweise auch von checkstyle[4] übernommen werden.

[3] http://www2.informatik.uni-wuerzburg.de/staff/fischer/tools/simple/
[4] http://checkstyle.sourceforge.net/

- **Kommentierungskonventionen:** Die Kommentierung entspricht den Konventionen von javadoc[5], mindestens sind die verlangten Tags in der gewünschten Reihenfolge vorhanden.

  Auch diese Anforderung ist über eine Konfigurationsdatei einstellbar. Unsinnige Kommentartexte selbst können oft mit einer einfachen Heuristik über Wort- und Buchstabenhäufigkeit entdeckt werden.

- **Spezifikation:** Das Programm entspricht strukturell der Spezifikation.

  Es wird überprüft, ob das Programm alle vorgeschriebenen Klassen, Attribute und Methoden enthält, ob die Vererbungshierarchie korrekt ist und ob die geforderten Ausnahmen geworfen werden.

  Um diese Überprüfung vornehmen zu können, muss die Schnittstelle des Programms relativ genau in der Aufgabenstellung angegeben sein. Der Vergleich wird mit einer Musterlösung durchgeführt, in der die geforderten Klassen, Attribute und Methoden durch bedeutungsvolle Kommentare markiert sind. Der Vergleich ist insofern tolerant, dass in der Vererbungshierarchie durchaus noch Zwischenklassen existieren dürfen.

- **Negativ-/Positiv-Liste:** Durch die Aufgabenstellung kann die Verwendung von gewissen Methoden, Klassen oder Paketen verboten oder erzwungen werden.

  Diese Negativ/Positiv-Liste wird dem Parser übergeben, der nun alle Importe, Klassensignaturen, Konstruktoren- und Methodenaufrufe, alle Variablenvereinbarungen usw. durchläuft und feststellt, ob die verbotenen Elemente gebraucht wurden. In diesem Fall wird das Programm nicht zugelassen und deshalb auch nicht geladen, was bei Überprüfung dieser Listen durch einen zugeschnittenen ClassLoader unvermeidlich wäre.

  Durch diesen Mechanismus können auch zusätzliche Sicherheitsaspekte realisiert werden, indem z.B. native Methoden oder alle Klassen aus `java.lang.reflect` verboten werden.

- **Funktionalitätstests:** Neben den strukturellen Überprüfungen wird die Funktionalität durch Tests abgeprüft.

  Auch dieses geschieht durch Vergleich mit der Musterlösung. Dazu wird entsprechend einer Testspezifikation jeweils eine Methode aus der Musterlösung, sowie die korrespondierende der eingegebenen Lösung aufgerufen und deren Ergebnisse verglichen. Man beachte, dass dieser Vergleich auf Objektebene und nicht auf der Stringebene der Ein- und Ausgabe erfolgt. Im Fehlerfall wird der Methodenname mit den Eingangswerten und dem erwarteten Ergebnis zurück gemeldet.

- **DejaGnu:** Blackbox-Tests können mit Hilfe von DejaGnu[6] durchgeführt werden. Hier wird für jeden Test zu einer Eingabe die korrekte Ausgabe erwartet und diese textuell verifiziert.

[5] http://java.sun.com/j2se/javadoc/
[6] http://www.gnu.org/software/dejagnu/

Einer Musterlösung kommt in unserem Ansatz eine hohe Bedeutung zu, sie dient ja nicht nur als Referenz, sondern als Vergleich und steuert auch noch einige wichtige Tests.

3.6 Variantengenerator

Um das Kopieren der Lösungen zu erschweren und um die Kommentierung fremder Lösungen zu ermöglichen, werden Programmieraufgaben parametrisiert. So können von einer Aufgabe bis zu 256 Varianten erzeugt werden. Für die Formulierung der Aufgabentexte ist dies relativ einfach mit dem Makroprozessor M4 möglich, für die Verwaltung und Generierung der Musterlösungen wurde ein Variantengenerator entwickelt.

Mit dessen Hilfe können alle Variationen einer Musterlösungsklasse in einer Datei zusammengefasst werden, aus der dann sämtliche Variationen mittels einer Präprozessierung erzeugt und kompiliert werden.

Die in den Varianten unterschiedlichen Quelltextstellen werden in Kommentare eingeschlossen, die in der M4 Syntax entsprechend der Aufgabenstellung formuliert sind. Dabei können auch Makros definiert werden.

Daraus wird nun für jede mögliche Variation der entsprechende kompilierbare Code erzeugt. Die erzeugten Variationen werden dann in einzelnen Verzeichnissen abgelegt. Pro Variation können natürlich mehrere Quelltexte (d.h. verschiedene Klassen) angegeben werden.

Um das Schreiben von Testmethoden zu erleichtern, kann der Variantengenerator automatisch Stub-Klassen erzeugen, die die Ergebnisse von Methodenaufrufen in der Musterlösung und der eingereichten Lösung vergleichen. Gleitkommazahlen gelten innerhalb eines vorgegebenen Differenzbereichs als gleich. Desweiteren wird überprüft, ob die gleichen Ausnahmen geworfen werden. Falls die Ergebnisse nicht übereinstimmen, werden entsprechende Meldungen auf den Ergebnisstrom geschrieben. Der Konstruktor einer Stub-Klasse benötigt die jeweiligen ClassLoader der beiden Lösungen sowie den Resultatsstrom, auf dem die Ergebnisse ausgegeben werden.

3.7 Ein Beispiel

Nachfolgend wird eine Musterlösung, die ja auch als Testspezifikation für einige der Prüfungen verwendet wird, vorgestellt.

Aufgabe ist es in einer Klasse `StringReverse` eine statische Methode `reverse` zu implementieren, die einen String als Parameter erhält, und diesen umgedreht zurückliefert. Dazu darf allerdings nicht die Methode `reverse` aus `StringBuffer` verwendet werden.

Die hier verwendete Musterlösung, in der auf `StringBuffer.reverse()` zurückgegriffen wird, dient in diesem Fall lediglich zur Prüfungsspezifikation. Sie hält sich der Einfachheit und Implementierungssicherheit halber selbst nicht an die Aufgabenstellung. Eine Weiter-

gabe an Studenten ist in diesem Fall nicht sinnvoll.

```
//-
package jopTest;

import de.uni_wuerzburg.informatik.praktomat.*

public class StringReverse {
    public static String reverse(String s) {
        if (s == null) return null;
        return new StringBuffer(s).reverse().toString();
    }

    //-
    public static void main(String[] args) { /* ... */ }

    //-
    public static void teste (BitSet bs, String group,
            GrayboxConstants con, FileResultStream fs,
            ClassLoader refCl, ClassLoader implCl) {
        try {
            stringReverse sr = new stringReverse(refCl, implCl, fs);
            sr.enableReturnValuesChecking(true);
            sr.reverse("X"); sr.reverse("nebel");
            sr.reverse("Würzburg mit Umlaut ...");
            sr.reverse(""); sr.reverse(null);
        } catch(Exception ex) {
            fs.addError("Fehler beim Testen: " + ex.toString());
        }
    }
}
```

Versucht die Teilnehmerlösung analog zur Musterlösung Methoden aus `StringBuffer` aufzurufen, so wird das durch die Überprüfung der extern definierten Negativ-Liste erkannt, und die Ausführung verhindert. Die Aufgabe ist dann natürlich nicht erfüllt:

> (-) Aufruf des Konstruktors 'java.lang.StringBuffer' in der verbotenen Klasse java.lang.StringBuffer
> (-) Aufruf der Methode 'toString' aus der verbotenen Klasse java.lang.StringBuffer
> (-) Aufruf der Methode 'reverse' aus der verbotenen Klasse java.lang.StringBuffer

Die erforderliche Struktur der Teilnehmerlösung wird durch den Vergleich mit der Struktur der Musterlösung geprüft. Nicht relevante Teile der Musterlösung, die also nicht geprüft werden müssen, werden durch spezielle Kommentare (`//-`) markiert.

In diesem Fall braucht also weder die Paketzugehörigkeit, noch die Methoden `main()` und `teste()` durch den Studenten erstellt werden. Insbesondere jedoch die Methode `reverse()` (mit gleicher Signatur) muss auch in der Teilnehmerlösung vorhanden sein.

Die eigentliche Funktionalität wird mit Hilfe der Methode `teste()` geprüft. Diese wird aufgefrufen und verwendet die bereitgestellten Stub-Klassen um einen Ablauf zu simulieren. Der Vergleich der Ergebnisse der Musterlösung und der Teilnehmerimplementierung wird dabei automatisch durch die Stub-Klassen erledigt, die dann auch das Gesamtergebnis und die Ausgabe entsprechend aufbereiten. Alternativ können die Vergleiche manuell programmiert werden. Dies erfordert allerdings wesentlich mehr Aufwand.

Gibt ein Teilnehmer z.B. folgendes Programm ab,

```
public class StringReverse {
    public static String reverse(String s) {
        if ( (s == null) || (s.length() <= 1) ) return s;
        return reverse(s.substring(1)) + s.charAt(1);
    }
}
```

so führt das zu folgender Fehlermeldung:

> (-) Teilnehmerimplementierung für public static java.lang.String StringReverse.reverse(java.lang.String) liefert **llebe** anstelle von **leben**
> (-) Teilnehmerimplementierung für public static java.lang.String StringReverse.reverse(java.lang.String) liefert **.... tualmU tim grubzrü** anstelle von **... tualmU tim grubzrüW**

Behebt man den Fehler (ersetze `s.charAt(1)` durch `s.charAt(0)`), so wird das Programm akzeptiert.

4 Neue Ansätze

4.1 Adaptivität und Benutzerführung

Durch eine tiefergehende Integration von Tutorial und Aufgaben könnte das System adaptiv auf die Fortschritte des Lernenden eingehen. Abhängig von dem Wissensstand des Lernenden, der durch die Verfolgung der gelesenen Lerneinheiten und deren Überprüfung durch die Aufgaben erfasst wird, ermittelt das Lernsystem die folgenden Lerneinheiten. Der Lernpfad wird also dynamisch den Fähigkeiten jedes einzelnen Benutzers angepasst.

Die Möglichkeit persönliche Notizen abzuspeichern erhöht die Individualisierung und sicher auch die Akzeptanz einer Lernumgebung beträchtlich.

4.2 JUnit

Erfahrungen mit JOP zeigten, dass es für die Studenten besser ist, wenn sie die Testmechanismen verstehen und die Wichtigkeit der Tests einsehen. Es sollte also auch das Testen gelehrt werden.

Als zusätzliche Prüfung wurde in den Praktomat die Fähigkeit integriert, JUnit[7] Testfälle auszuführen. Die Prüfung kann die komplexen Funktionalitätstests basierend auf dem Musterlösungvergleich ersetzen, eine Simulation der Ein-/Ausgabe ist so allerdings nicht möglich.

[7] http://www.junit.org/

Soll auch das Testen Inhalt der Lehre sein, so gibt der Student in Zukunft nicht nur ein Programm ab, sondern auch eine Reihe von Testprogrammen, die dann vom Praktomat aufgerufen werden. Dann sollten die Qualität und die Vollständigkeit der Testfälle überprüft werden.

4.3 XJET

Um den Prozess der Aufgabenstellung einschließlich Testspezifikation und Bereitstellen von Testprogrammen zu steuern, wird zur Zeit das Werkzeug XJET (XML Java Exercises and Tests) entwickelt. Die Aufgabenstellung und die Spezifikation der Testfälle werden dabei in einem XML Dokument zusammengefasst. Damit ist klar, dass zur Aufgabenstellung auch die Bescheibung der durchzuführenden Tests gehört, diese müssen nicht mehr über den Umweg einer aufwändigen Musterlösung implementiert werden. Da man auch den strukturellen Test der Schnittstelle und der Vererbungshierarchie recht einfach auf Grund von Markierungen im XML Aufgabentext durchführen kann, wird die Musterlösung nicht mehr benötigt.

4.4 Diskussion

Durch den Praktomat als zentrale Verwaltungsschnittstelle für Aufgaben, Musterlösungen und Teilnehmerlösungen wurde ein organisatorischer Rahmen geschaffen, der sowohl die Studenten als auch die Korrektoren stark unterstützt. Es wurde gezeigt, dass weitgehende Überprüfungen von Java Programmen automatisch durchgeführt werden können. Durch genaue Aufgabenstellung wurden den Studenten ein vernünftiger Programmaufbau abverlangt, weitere „Formalien“ erhöhen die Qualität des Quelltextes beträchtlich.

Die mit der Musterlösung durchgeführten Vergleiche decken bei sorgfältiger Programmierung alle relevanten Fälle ab. Das Schreiben der Musterlösung ist allerdings insbesondere bei Verwendung von Varianten deutlich aufwändiger als bisher.

5 Evaluation

JOP und insbesondere der “Java-Praktomat” wurde in verschiedenen Vorlesungen und Praktika mit Erfolg eingesetzt. Es wurden alleine in zwei Praktika in einem Semester ca. 2500 Programme für 16 unterschiedliche Aufgaben eingereicht und beurteilt. Die durchschnittliche Größe der Programme lag bei etwa 1000 Zeilen.

Die Existenz der Testfälle wurde allgemein begrüßt. Die Studenten erhielten die Gelegenheit ihre Entwürfe vom Praktomat beliebig oft testen zu lassen, wovon viele auch ausgiebig Gebrauch machten.

Die Verlässlichkeit der Tests beweist die Tatsache, dass nur 2% der vom Praktomat akzep-

tierten Lösungen, von dem Schlusskorrektor als nicht ausreichend eingestuft wurden.

Der Arbeitsaufwand für die Beurteilung konnte durch den einheitlich formatierten, gut lesbaren und gut kommentierten Quellcode sowie durch die Ausführung der Tests etwa um einen Faktor von 4 vermindert werden. Ja, nur durch den Einsatz dieser Hilfsmittel war eine gerechte und sorgfältige Beurteilung aller Teilnehmer überhaupt erst möglich.

Es ging jedoch nicht nur um die Bewertung. Es ist ausdrücklich zu bemerken, dass die Qualität der abgegebenen Programme gegenüber früheren Praktika deutlich verbessert wurde, wobei die Durchfallquote etwa konstant blieb. Wir glauben, dass die Studenten den gelernten, professionellen Programmierstil auch beibehalten werden.

Danksagung

Das JOP Projekt wurde an den Universitäten Passau und Würzburg durchgeführt und von der virtuellen Hochschule Bayern (vhb) gefördert.

Literatur

[1] JOP; `http://jop.informatik.uni-wuerzburg.de/`

[2] J. Dockx, E. Steegmans: *A New Pedagogy for Programming*, Sixth Workshop on Pedagogies and Tools for Learning Object Oriented Concepts, ECOOP 2002, `http://prog.vub.ac.be/ecoop2002/ws03/`

[3] H. Eichelberger: *JTransform - A Java Source Code Transformation Framework* TR 303, Institute for Computer Science, Würzburg University, 2002

[4] G. Fischer, J. Wolff v. Gudenberg: *Java Online Pedagogy*, Seventh Workshop on Pedagogies and Tools for Learning Object Oriented Concepts, ECOOP 2003

[5] G. Fischer, J. Wolff v. Gudenberg: *JOP, ein Java Online Praktikum der vhb*, Softwaretechnik-Trends, 22(4), Nov 2002, p. 15-18

[6] G. Görlitz, S. Müller: *Didaktisches Design für eine Online-Programmierausbildung*, Softwaretechnik-Trends 22:3, S.32-35, 2002

[7] J. Krinke, M. Störzer, A. Zeller: *Web-basierte Programmierpraktika mit Praktomat*, Softwaretechnik-Trends 22:3, S.51-53, 2002

[8] R. Lütticke, C. Gnörrlich, H. Helbig: *Internet-basierte Lehre im Rahmen von VILAB*, in Schubert, Reusch, Jesse (Hrsg.): Informatik bewegt, GI-Edition, Lecture Notes in Informatics, P-19, 2002

[9] C. Sauer: *Die Verständlichkeit von Texten, Visualisierungen und Bildschimen*, in Jakobs (Hrsg.): Textproduktion, Lang, S. 92-109, 1999

[10] R. Schulmeister: *Grundlagen hypermedialer Lernsysteme*, Oldenbourg, 2002

[11] A. Zeller: *Funktionell und verständlich programmieren - so lernen es die Passauer*, Softwaretechnik-Trends 19:3, Nov. 1999