WARP – ein Trainingssystem für UML-Aktivitätsdiagramme mit mehrschichtigem Feedback

Marianus Ifland, Felix Herrmann, Julian Ott, Frank Puppe

Lehrstuhl für Informatik VI, Universität Würzburg
{ifland|f.hermann|ott|puppe}@informatik.uni-wuerzburg.de

Abstract: Die UML-Aktivitätsdiagramm-Modellierung ist ein wichtiges Thema im Informatik-Studium und verwandten Studiengängen und dient als Grundlage der Programmierausbildung. Studierende sollen dabei lernen, bereits vor der Implementierung über ihre Programme nachzudenken und diese auf einem höheren Abstraktionsniveau zu skizzieren. Aus pragmatischen Gründen werden entsprechende Übungsaufgaben in der Regel separat gestaltet, so dass entweder die abstrakte Modellierung oder die Programmierung geübt und bewertet wird. Wir haben nun einen Aufgabentyp konzipiert und eine entsprechende tutorielle Web-Anwendung WARP implementiert, mit der die Modellierung von dynamischen Programmeigenschaften mit Hilfe von Aktivitätsdiagrammen gemeinsam mit der Erstellung von Programmcode eingeübt werden kann. Studierende zeichnen dabei Aktivitätsdiagramme, deren Syntax um konkreten Programmcode erweitert ist, und definieren so das Verhalten eines Agenten in einer Roboter-Simulations-Umgebung, der dort eine Aufgabe zu lösen hat. Die Studierenden erhalten zu ihren gezeichneten Aktivitätsdiagrammen Feedback auf mehreren Ebenen: 1) syntaktische Korrektheit des Diagramms, 2) syntaktische Korrektheit des automatisch erzeugten Programmcodes, 3) visuelle Rückmeldung der Ausführung des Programms in einer Robotersimulationsumgebung, 4) Unit-Tests bzw. Metriken zur Ausführung des Programmes. Insbesondere das visuelle Feedback trägt nicht nur zur Motivationssteigerung bei, sondern regt auch zum Nachdenken und Verstehen der Programmkonstrukte an. Die Anwendung wurde im Übungsbetrieb einer Softwaretechnik-Vorlesung mit etwa 250 Studierenden im Sommersemester 2013 eingesetzt und erfolgreich evaluiert.

1 Einleitung

Die Programmierausbildung ist ein zentrales Element im Informatik-Studium und informatiknaher Studiengänge. Dabei soll auch ein Verständnis von Programmen auf höheren Abstraktionsebenen vermittelt werden. Dafür eignet sich die UML-Notation [RJB04] [Ba04]. Mit Klassendiagrammen wird die Datenstruktur übersichtlich dargestellt, mit Aktivitätsdiagrammen kann die Programmlogik sowohl auf abstrakter als auch auf konkreter, ausführbarer Ebene modelliert werden. Bei geeigneter Modellierung und Anordnung der Elemente kann die visuelle Darstellung dabei das Verständnis erleichtern und von syntaktischen Details der Programmiersprache abstrahieren. Daher sind Aktivitätsdiagramme ein gutes didaktisches Mittel, um das Verständnis von großen, aber auch kleinen Programmen, wie sie in Anfängerlehrveranstaltungen genutzt werden, zu fördern.

Beim Einsatz für Übungsaufgaben sollte den Studierenden ein qualitativ hochwertiges Feedback gegeben werden, was mit manueller Korrektur bei vielen Studierenden aufwändig ist. Für Strukturdiagramme, beispielsweise Klassendiagramme, existieren bereits Trainingssysteme, die von Studierenden erstellte Diagramme automatisch bewerten und entsprechendes Feedback generieren [ASI07][HQW08][If12][SG11]. In diesem Beitrag stellen wir nun ein neuartiges Trainingssystem vor, in dem aus Aktivitätsdiagrammen ausführbarer Code für eine Robotersimulationsumgebung erzeugt und ein entsprechendes visuelles Feedback generiert wird. Dadurch wird ein Anreiz geschaffen, präzise Aktivitätsdiagramme zu schreiben, ohne vollständig auf die Ebene einer Programmiersprache gehen zu müssen. Das Trainingssystems WARP (Würzburger Aktivitätsdiagramm Roboter Programmierung) soll dabei helfen, die folgenden Lernziele zu erreichen. Zum einen sollen Anwender das gewünschte Verhalten eines Roboters als syntaktisch korrektes Aktivitätsdiagramm abbilden können. Dies schließt das Verständnis der Kernelemente von Programmen (Aufrufe von Prozeduren, Verzweigungen, Schleifen etc.) mit ein. Zum anderen soll das Verständnis des Zusammenhangs von grafischen Aktivitätsdiagrammen und textuellem Programmcode verbessert werden.
In dem Trainingssystem wird derzeit nicht der Aufbau von Datenstrukturen geübt, d. h. die Kopplung zu Klassendiagrammen fehlt (noch). Stattdessen werden die Datenstrukturen bzw. Klassen in den Übungsaufgaben vorgegeben.

2 Methoden

Das Trainingssystem stellt Aufgaben im Kontext einer Robotersimulationsumgebung. Primitive Roboterbefehle, wie beispielsweise `moveForward()`, `rotateRight()`, `rotateLeft()`, `markCurrentNode(Color c)` sind vorgegeben. Die Studierenden lösen die Aufgabe durch Zeichnen eines Aktivitätsdiagramms in der tutoriellen Web-Anwendung WARP. Zwei unterschiedliche Lösungen für die Aufgabe „Programmiere einen Roboter so, dass er ein Rechteck mit variabler Breite und Höhe zeichnet“ zeigen Abb. 1. und Abb. 2. Insbesondere werden in WARP auch strukturierte Knoten für Schleifen und Verzweigungen der UML unterstützt. Nach Eingabe des Diagramms erhalten Studierende Feedback auf vier Ebenen:

- **Erste Feedback-Ebene**: Falls das Diagramm syntaktisch nicht korrekt ist, werden Fehler direkt im Diagramm mit einem textuellen Kommentar angezeigt (Abb. 3).
- **Zweite Feedback-Ebene**: Falls der aus dem Diagramm automatisch erzeugte Java-Code syntaktische Fehler hat, werden entsprechende Hinweise generiert.
- **Dritte Feedback-Ebene**: Der Java-Code wird in der Robotersimulationsumgebung RoSE [He13] ausgeführt und es wird eine Animation erzeugt, welche die Studierenden schrittweise abspielen können.
- **Vierte Feedback-Ebene**: Die Studierenden erhalten eine Rückmeldung, ob die Aufgabe korrekt gelöst wurde. Zudem werden Metriken über primitive Roboteraktionen angezeigt, z. B. wie viele Schritte der Roboter gebraucht hat.

Bei Fehlermeldungen können die Studierenden das Aktivitätsdiagramm abändern und die aus dem automatisch erzeugten Java-Code generierte neue Animation ansehen.

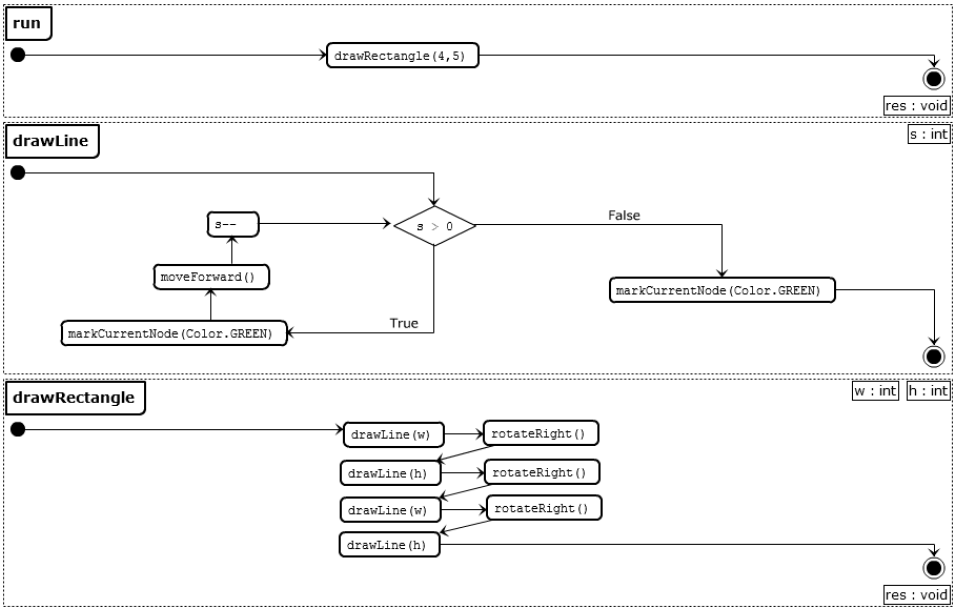

Abbildung 1: Im WARP-Editor gezeichnetes, aus drei Aktivitäten bestehendes Aktivitätsdiagramm zum Zeichnen eines Rechtecks für eine Robotersimulationsumgebung.

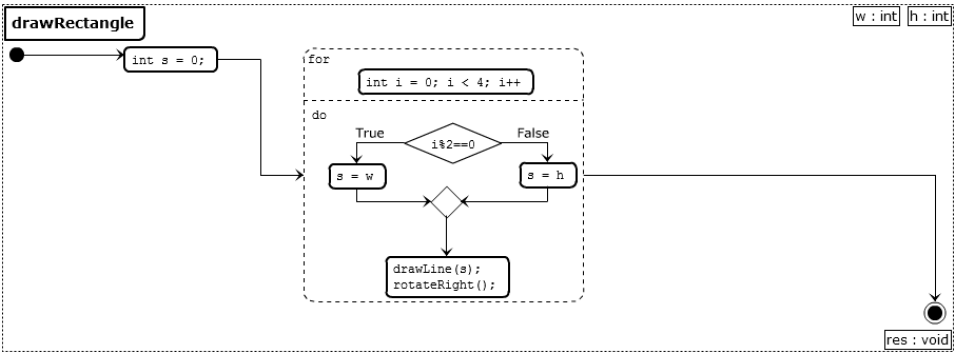

Abbildung 2: Alternative Lösung für die Aktivität `drawRectangle` aus Abb. 1. Die Aktivitäten `run` und `drawLine` bleiben gleich und werden hier nicht erneut dargestellt.

2.1 Erste Feedback-Ebene: Syntax des Aktivitätsdiagramms

Ein syntaktisch korrektes Aktivitätsdiagramm in WARP besteht aus einer Menge von Aktivitäten. Eine Aktivität besteht aus ihrer Signatur, einem Rückgabe-Parameter und einem gerichteten Graphen. Eine Signatur hat die gleiche Syntax wie die Signatur einer Java-Methode.

Der gerichtete Graph in einer Aktivität ist ein Paar mit einer Menge von Knoten bestimmten Typs und einer Menge von Kanten. Die Knotentypen sind: Startknoten, Endknoten, Aktionsknoten, Verzweigungsknoten, Verbindungsknoten, If-Then-Knoten, For-Do-Knoten, Do-While-Knoten und While-Do-Knoten. Jeder gerichtete Graph muss

dabei genau einen Start- und genau einen Endknoten enthalten. Ein- und Ausgangsgrad jeden Knotens ist 1, wobei folgende Ausnahmen gelten: Der Eingangsgrad von Startknoten und der Ausgangsgrad von Endknoten ist 0. Außerdem ist der Eingangsgrad von Verzweigungs- und Verbindungsknoten sowie der Ausgangsgrad von Verzweigungsknoten 1 oder 2.
Aktionsknoten und Verzweigungsknoten beinhalten Text, wobei dieser auch leer sein kann. If-Then-, For-Do-, While-Do- und Do-While-Knoten bestehen aus einer Liste von Blöcken. Es existieren folgende Typen von Blöcken: If-Block, Then-Block, Else-If-Block, For-Block, While-Block und Do-Block. Welche Knoten welche Blöcke in welcher Reihenfolge enthalten können, ergibt sich trivial aus der Java-Syntax der entsprechenden Kontrollstrukturen. Folgende Blöcke beinhalten einen einzelnen Aktionsknoten: If-Block und Else-If-Block (in If-Then-Knoten), For-Block (in For-Do-Knoten) und While-Block (in While-Do- und Do-While-Knoten). Then-Blöcke (in If-Then-Knoten) und Do-Blöcke (in For-Do-, While Do- und Do-While-Knoten) werden als Knoten-Container bezeichnet und beinhalten einen gerichteten Graphen, für den die gleichen syntaktischen Regeln gelten, wie für den Graphen einer Aktivität. Ausnahme ist, dass sie keine Start- und Endknoten besitzen. Stattdessen müssen sie genau einen Knoten ohne eingehende und genau einen Knoten ohne ausgehende Verbindung enthalten. Diese Knoten dienen dann als implizite Start- und Endknoten dieses reduzierten Graphen.
Da diese Anwendung für Studienanfänger konzipiert wurde, wurde auf einige Konzepte aus der UML2 verzichtet, beispielsweise auf Parallelisierung (Splitting und Synchronisation), Objektknoten, Ausnahmebehandlung und Ereignisse. Des Weiteren wurde die Syntax präzisiert, um die Aktivitätsdiagramme maschinell weiter verarbeiten zu können. Bei reduzierten Graphen innerhalb von Knoten-Containern wurde auf explizite Start- und Endknoten zugunsten einer besseren Übersicht beim Zeichnen der Graphen verzichtet.

Bei der Eingabe eines Aktivitätsdiagramms in WARP haben Studierende die Möglichkeit, die syntaktische Korrektheit ihres Diagramms zu prüfen und sich entsprechende Fehlermeldungen anzeigen zu lassen. Ein Beispiel zeigt Abbildung 3.

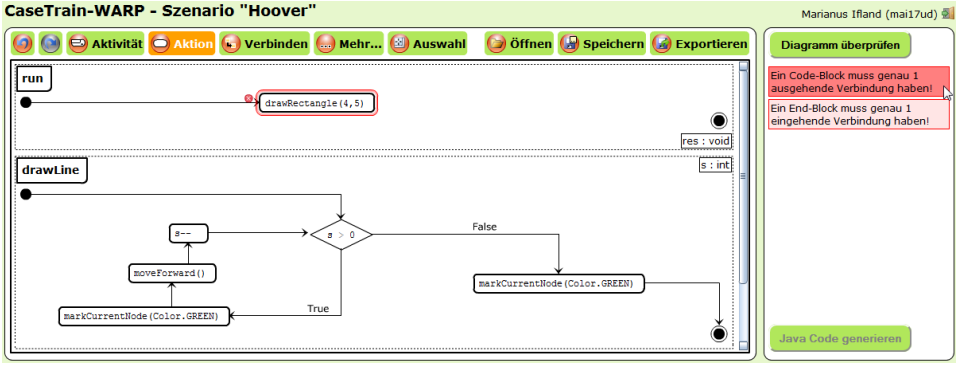

Abbildung 3: Erste Feedback-Ebene: Rückmeldung zur Syntax des Aktivitätsdiagramms. Beim Überfahren der Fehlermeldung mit dem Mauszeiger wird die entsprechende fehlerhafte Stelle im Diagramm farblich hervorgehoben.

2.2 Zweite Feedback-Ebene: Syntax des generierten Java-Codes

Ist das Aktivitätsdiagramm syntaktisch korrekt, wird daraus Java-Code generiert. Jede Aktivität des Diagramms wird dabei in eine Methode einer Java-Klasse übersetzt und in eine Klassenschablone eingesetzt.

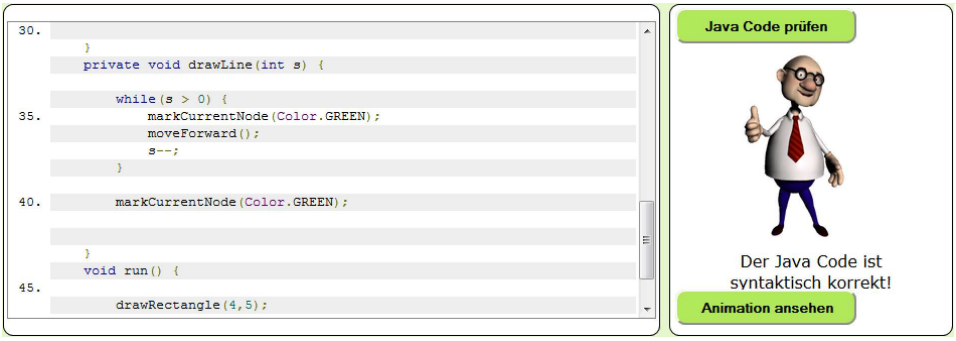

Abbildung 4: Positives Feedback zur Syntax des generierten Java-Codes

Diese Klasse erbt dabei von einer speziellen RoSE-Klasse, welche auch die primitiven Roboterbefehle zu Verfügung stellt. Anschließend wird der Java-Code auf syntaktische Korrektheit geprüft. Analog zur vorigen Feedback-Ebene wird dabei eine Liste der Fehlermeldungen angezeigt. Auch hier wird beim Überfahren der Fehlermeldung mit dem Mauszeiger die entsprechende fehlerhafte Stelle im Java-Code farblich hervorgehoben. Die Fehlermeldungen werden dabei direkt von einem Java-Compiler erzeugt. Abb. 4 zeigt den Java-Code zur Aktivität `drawLine` aus Abb 2.

2.3 Dritte Feedback-Ebene: Semantik des Java-Codes (Animation)

Bei syntaktischer Korrektheit kann das Programm ausgeführt werden, wobei serverseitig eine RoSE-Animation erstellt wird. RoSE instanziiert dazu die entsprechende Aufgabe (z. B. „Rechteck“) und ruft die Methode `run()` des erstellten Roboters auf.

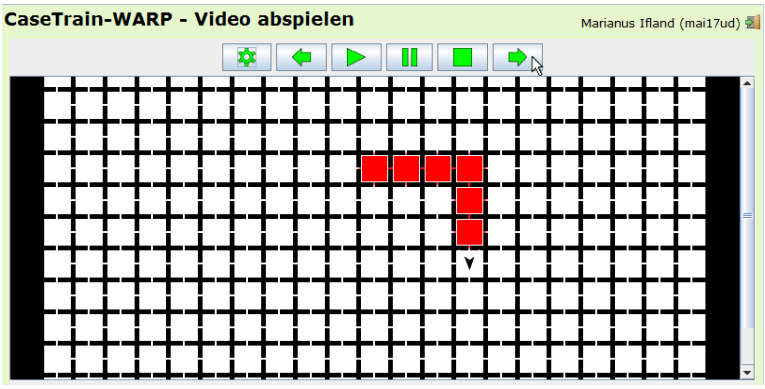

Abbildung 5: Dritte Feedback-Ebene: In einer Animation sehen die Studierenden, wie sich der von ihnen selbst erstellte Roboter in einer Simulationsumgebung verhält. Hier besteht die Aufgabe darin, durch das Markieren von Feldern ein Rechteck bestimmter Seitenlängen zu zeichnen.

Der Roboter verändert nun schrittweise den Zustand der Umgebung (z. B. durch Einfärben eines Feldes) und den eigenen Zustand (z. B. durch Veränderung der Position). Diese Zustände der Umgebung und des Roboters werden in einem internen Format gespeichert und ergeben als Sequenz die RoSE-Animation. Diese kann nun im RoSE-Player betrachtet werden, entweder durch explizites Aufrufen des nächsten Schrittes oder als laufende Animation (Abb. 5). Die Studierenden können so sehen, ob der erstellte Roboter sich korrekt, also entsprechend der Aufgabe verhält und diese löst oder nicht.

2.4 Vierte Feedback-Ebene: Semantik des Java-Codes (Erfolg und Metriken)

Zusätzlich zur Animation erhalten Studierende noch eine textuelle Rückmeldung darüber, ob der erstellte Roboter das in der Aufgabe definierte Ziel erfüllt hat, die Aufgabe also bestanden wurde. Außerdem können hier je nach Aufgabe weitere Meldungen über die Ausführung des Programmes gegeben werden. Beispielsweise wird meist angezeigt, in wie vielen Schritten die Aufgabe vom programmierten Roboter gelöst wurde.

3 Implementierung

3.1 Grundlegendes Konzept

WARP wurde weitgehend als Single-Page-Web-Anwendung implementiert, was bedeutet, dass sie aus einem einzelnen HTML-Dokument besteht (Abb. 6). Neue Inhalte werden dynamisch nachgeladen, ohne dass die komplette Seite neu geladen und vom Browser neu gerendert werden muss. Ausnahme bildet der Dialog zur Betrachtung des Verhaltens des erstellten Roboters in der virtuellen Umgebung (Abb. 5), welcher auf einer separaten Seite angezeigt wird. Aktivitätsdiagramm und zugehöriger Programmcode werden den Studierenden also stets gemeinsam angezeigt. WARP wurde aus Gründen der Usability für herkömmliche Rechner (also nicht für mobile Endgeräte) konzipiert.

Abbildung 6: Vier Bereiche im Hauptdialog der WARP Web-Anwendung

3.2 Anbindung an Moodle und Authentifizierung

Die Webanwendung wird von der Lernplattform Moodle[1] aufgerufen, wobei diese die Benutzerkennung, die Kennung der zu bearbeitenden Aufgabe und einen persönlichen Authentifizierungsschlüssel sendet. Dieser Authentifizierungsschlüssel wird von Moodle aus IP-Adresse des Benutzers, Benutzerkennung und einem statischen Schlüsselwort generiert. Der Webanwendung sind Schlüsselwort und Algorithmus zur Erstellung des Authentifizierungsschlüssels bekannt. Beim Aufruf von WARP wird aus übermittelter Benutzerkennung und der IP-Adresse des Benutzers ebenfalls ein Authentifizierungsschlüssel berechnet und mit dem übermittelten verglichen. Sind die Authentifizierungsschlüssel identisch, ist der Benutzer authentifiziert. Andernfalls wird die Anmeldung abgewiesen. Mit diesem Ansatz benötigen die Studierenden also keine zusätzlichen Zugangsdaten.

3.3 Verwendete Technologien

Die verwendeten Technologien sind clientseitig Adobe Flash, HTML5, JavaScript und CSS3. Adobe Flash wird für den Aktivitätsdiagramm-Editor verwendet, welcher ursprünglich für den Flash-basierten CaseTrain-Player [Hö09] entwickelt wurde. Gegenwärtig wird der Editor für WARP im Rahmen einer Neuimplementierung in HTML5 und JavaScript umgesetzt. In der hier vorgestellten und evaluierten Version von WARP wurde zur Anzeige der Animation ein Java-Applet verwendet, da so der Standard-Player des RoSE Systems einfach eingebunden werden konnte. Für eine aktualisierte Version von WARP wurde inzwischen ein eigener Player mittels HTML5 und JavaScript implementiert.
Um RoSE und weitere benötigte Bibliotheken einfach einbinden zu können, wird serverseitig Java7 auf Apache Tomcat 6 und MySQL5 verwendet.

3.4 Kommunikation

Kommunikation zwischen Client und Server findet mittels AJAX und JSON zwischen JavaScript und Java-Servlets statt. Zur Übermittlung des Aktivitätsdiagramms an den Server wird dieses in ein XML-Dokument konvertiert. Dieses wird serverseitig zur weiteren Verarbeitung in ein Java-Objekt umgewandelt, wobei die „Java Architecture for XML Binding“ (JAXB) verwendet wird. Der serverseitig generierte Java-Code wird in Textform, die ebenfalls serverseitig generierte RoSE Animation im JSON-Format an den Client geschickt.

3.5 Sicherheit

Benutzer der Anwendung erstellen clientseitig Programmcode, welcher serverseitig kompiliert und ausgeführt wird, um die Animation zu erzeugen. Es besteht also die Gefahr, dass über die Webanwendung absichtlich oder unabsichtlich Schadcode auf den Server

[1] https://moodle.org/ (März 2014)

gespielt und dort ausgeführt wird. Eine Gefahr ist dabei lesender oder schreibender Zugriff auf das Dateisystem des Servers. Dies wird in WARP auf zweierlei Arten verhindert. Zum einen läuft der Tomcat-Prozess unter einem gesonderten Account, der keine Administratorenrechte hat und dessen Zugriff auf das Dateisystem auf bestimmte Ordner beschränkt ist. Des Weiteren wird der Java-Code der Benutzer in einer separaten virtuellen Maschine ausgeführt, deren Zugriffsrechte mit Hilfe des Security-Managers und einem übergebenen Policy-File sehr stark eingeschränkt sind. Dieser Ansatz wird in ähnlichen Systemen ebenso verwendet [HQW08].
Eine weitere Gefahr beim Ausführen von Fremdcode besteht darin, dass beispielsweise durch Endlosschleifen zu viele Systemressourcen verwendet werden, womit diese für andere Prozesse nicht mehr zur Verfügung stehen. Dies wird in WARP bzw. in RoSE selbst dadurch verhindert, dass die Ausführung des Programms nach einer gewissen Rechenzeit abgebrochen wird.

3.6 Übersetzung des Aktivitätsdiagramms in Java-Code

Ein Aktivitätsdiagramm wird stets in eine Java-Klasse übersetzt, wobei die Aktivitäten Methoden dieser Klasse entsprechen. Wie in Abschnitt 2.2 beschrieben, ist eine Klassenschablone vorgegeben, in welche die Methoden eingesetzt werden. Die Signatur der Methode wird direkt aus der Signatur der Aktivität übernommen. Der Methodenrumpf wird über den gerichteten Graphen der Aktivität definiert. Der Graph wird dabei zunächst in einen abstrakten Syntaxbaum überführt. Bei einem abstrakten Syntaxbaum handelt es sich um die Repräsentation von Quellcode als Baumgraph. Dessen Knoten repräsentieren dabei bestimmte Sprachkonstrukte, die Kanten deren Zusammensetzung. Abstrakt ist ein solcher Baum deshalb, weil bestimmte Ausdrücke der Programmiersprache nicht explizit vorkommen, sondern durch die Knoten implizit angegeben sind.
Die in WARP generierten Syntaxbäume sind nur teilweise abstrakt, da es in den Graphen der Aktivitäten Aktionsknoten gibt, in denen konkreter Java-Code steht. Dieser Code wird bei der Erstellung des Baumes nicht weiter konvertiert, sondern durch einen einzelnen „konkreten“ Knoten repräsentiert (Abb. 7).

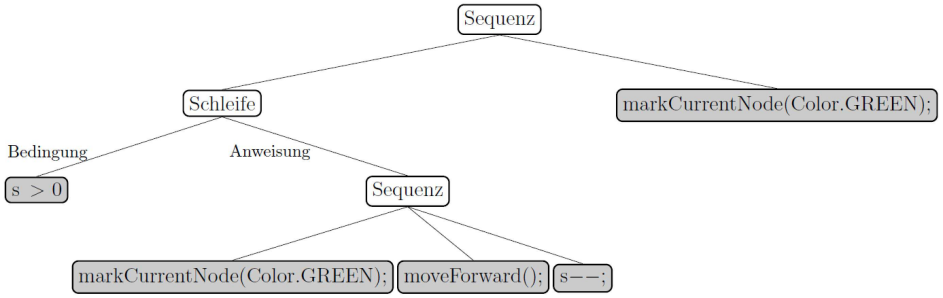

Abbildung 7: Zur Umwandlung in Java-Code wird aus dem gerichteten Graphen einer Aktivität zunächst ein teilweise abstrakter Syntaxbaum mit abstrakten (weiß) und konkreten Knoten (grau) erstellt, der anschließend in Java-Code übersetzt wird. Die Abbildung zeigt den teilweise abstrakten Syntaxbaum der Aktivität `drawLine` des Aktivitätsdiagramms in Abb. 1.

Die Wurzel des Syntaxbaumes ist stets eine „Sequenz“. Weitere Knoten im Baum ergeben sich aus den im Aktivitätsdiagramm verwendeten Knoten: While-Do-, Do-While-, und

For-Do-Knoten erzeugen „Schleifen“ (unterschiedlichen Subtyps), If-Then-Knoten und Verzweigungs- /Verbindungsknoten erzeugen „Verzweigungen“ oder „Schleifen“. Der Syntaxbaum wird nun in Java-Code umgewandelt, wobei das *Java Code model* verwendet wird. Das Umwandeln eines Teilbaums in Java-Code geschieht, indem abhängig von der Wurzel des Teilbaums eine bestimmte Statement-Schablone“ erstellt wird. Ein „Schleife“-Knoten ergibt beispielsweise die Schablone:

```
while(linker Teilbaum){
    rechter Teilbaum
}
```

Durch rekursives Aufrufen der Routine für die Teilbäume, dessen Wurzeln die Kinder des aktuellen Knotens sind, wird diese Schablone nun gefüllt. Die Rekursion endet, wenn ein Teilbaum nur aus einem Blatt besteht. Blätter eines Syntaxbaums sind stets konkrete Knoten.

3.7 Datenhaltung

WARP speichert jede Bearbeitung in einer MySQL-Datenbank ab. Dabei werden die Benutzerkennung, das Aktivitätsdiagramm und ggfs. der erzeugte Java-Code und die erzeugte RoSE-Animation sowie weitere Metadaten zur Bearbeitung gespeichert. Diese Daten ermöglichen eine umfassende statistische Auswertung.

4 Ergebnisse

Das Trainingssystem wurde im Sommersemester 2013 in den Übungen zur Vorlesung Softwaretechnik zur Erstellung von Aktivitätsdiagrammen eingesetzt. Die Gruppe der Studierenden war dabei heterogen, da die Vorlesung von verschiedenen Studiengängen besucht wird: Studierende der Informatik, der Luft- und Raumfahrtinformatik, der Wirtschaftsinformatik, der Wirtschaftsmathematik, der Mensch-Computer-Systeme und anderen, die meist im ersten oder zweiten Semester waren. Dabei wurden folgende Aufgaben gestellt:

- Aufgabe „Rechteck“: „Zeichnen Sie ein Rechteck mit den Seitenlänge 4 und 5“
- Aufgabe „Hoover“: „Markieren Sie alle Felder in einer beliebigen Farbe“
- Aufgabe „Labyrinth“: „Finden Sie das farblich markierte Feld“

Die Aufgaben waren dabei ausführlicher und bildhafter formuliert als hier angegeben. Die vollständige Aufgabenstellung der Aufgabe „Labyrinth“ lautete beispielsweise:

> „Definieren Sie in einem Aktivitätsdiagramm eine generische Methode `findeSchatz()`, die in einem unbekannten Labyrinth einen Schatz findet. Der Schatz liegt auf einem farbigen Einzelfeld, das Sie mit der Methode `isMarked()` entdecken, d. h. `isMarked()` liefert auf dem Feld, auf dem sich der Schatz befindet, `true` zurück und sonst `false`. In der Ausgangssituation steht der Roboter in der linken oberen Ecke des Feldes und schaut nach rechts.“

Die Bedienung des WARP-Editors wurde den Studierenden in einer Übungsstunde vorgeführt. Die Aufgaben wurden bewertet, wobei die Studierenden beliebig oft Programme eingeben durften. In früheren Semestern wurden ebenfalls Aufgaben zu

Aktivitätsdiagrammen gestellt, wobei die Lösungen als Dokument über eine Lernplattform oder auf Papier abgegeben und manuell von studentischen Hilfskräften korrigiert und bewertet wurden.
Die Studierenden konnten alle Aufgaben innerhalb eines Zeitraumes von einer Woche beliebig oft bearbeiten. WARP speicherte während dieses Experiments jedes Aktivitätsdiagramm nach jeder Änderung in einer Datenbank ab, sofern es syntaktisch korrekt und auch der generierte Java-Code syntaktisch korrekt war. Es gab also keine expliziten Einreichungen. Auf diese Weise wurden über 12.000 Aktivitätsdiagramme von 255 verschiedenen Studierenden in der Datenbank gespeichert. Daraus wurde nun für jede Aufgabe eine Teilmenge gebildet, welche jeweils die „finalen Bearbeitungen“ der Studierenden für diese Aufgabe beinhaltet. Die finale Bearbeitung eines Studierenden bei einer Aufgabe ist das zuletzt erstellte Diagramm, welches zu einer erfolgreichen Lösung führte bzw. bei Studierenden, die die Aufgabe nicht lösen konnten, das zuletzt erstellte Diagramm. Die Teilmengen beinhalten also für jeden Studierenden genau eine Bearbeitung pro Aufgabe. Tab. 1 zeigt Anzahl und Erfolgsquote der Bearbeitungen sowie Statistiken über die Verwendung bestimmter Konstrukte in den Aktivitätsdiagrammen.

	Rechteck	**Hoover**	**Labyrinth**	**gesamt**
Anzahl Bearbeitungen	247	229	209	685
Anteil erfolgreicher Bearbeitungen	92,9%	92,2%	87,0%	90,4%
Ø Anzahl Aktionsknoten	9,1	15,7	8,9	11,2
Ø Anzahl Verzweigungs-/Verbindungsknoten	1,4	1,1	1,9	1,4
Ø Anzahl If-Then-Knoten	0,2	1,2	1,0	0,8
Ø Anzahl Schleifenknoten	2,2	2,4	1,1	2,0

Tabelle 1: Nutzungsstatistik im Sommersemester 2013

Eine mögliche Erklärung für den hohen Anteil erfolgreicher Bearbeitungen ist, dass die Aufgaben innerhalb des Bearbeitungszeitraums beliebig oft wiederholt werden konnten und bei jedem Versuch entsprechendes Feedback gegeben wurde, das den Studierenden geholfen hat, schrittweise korrekte Lösungen zu entwickeln.
Es wurde untersucht, ob die Verwendung bestimmter Konstrukte einen Einfluss auf erfolgreiche Bearbeitungen hatte. Dabei wurde festgestellt, dass Studierende, die strukturierte If-Then-Knoten nutzten, einen Anteil an erfolgreichen Bearbeitungen von 91,8% hatten, Studierende dagegen, die keine strukturierten If-Then-Knoten, sondern Verzweigungsknoten (Raute) nutzten, einen Anteil von nur 78,8%. Dieser Unterschied ist signifikant auf einem Niveau von 1% (Wilcoxon-Mann-Whitney-Test). Eine mögliche Erklärung ist, dass Studierende mit besseren Programmierkenntnissen lieber strukturierte If-Then-Knoten verwenden, da diese sehr ähnlich zu konkreter Java-Syntax sind.
Stichprobenartige Überprüfungen ergaben, dass einige Studierende bei der Aufgabe „Labyrinth“ keine allgemeine Lösung für eine Labyrinth-Umgebung entworfen, sondern statt dessen auf das vorhandene, statische Labyrinth hin optimiert haben.

Am Ende der Vorlesung wurde eine Befragung der Studierenden zur Nutzung von WARP und anderer Trainingssysteme für andere Aufgabentypen durchgeführt, deren Ergebnisse in Tab. 2 dargestellt werden.

Aspekt	**Ø Note**
Konzept des Tools, unabhängig von der Implementierung	2,25
Umsetzung des Tools, Gesamtnote	2,61
Aspekt Bedienbarkeit (Einarbeitung, intuitive Bedienbarkeit)	2,47
Aspekt Technik (Verfügbarkeit, Robustheit, Effizienz)	3,57
Aspekt Inhalt der Übungsaufgaben	2,18
Aspekt Fairness der Bewertung der Übungsaufgaben	2,59
Aspekt hilfreiche Erklärungen als Test oder nützliches Feedback	2,95

Tabelle 2: Verschiedenen Aspekte von WARP wurden in einer Umfrage mit Schulnoten (1-6) bewertet (n = 240)

In der gleichen Befragung wurde zudem die Frage gestellt, ob die Studierenden es für sinnvoll halten, elektronische Tools (u.a. WARP) im Übungsbetrieb in Zukunft weiter einzusetzen. Diese Frage wurde von 93% der Studierenden mit „Ja“ beantwortet.

5 Diskussion und Ausblick

Aufgrund des durchgeführten Experiments lässt sich sagen, dass WARP effektiv bedienbar ist. Die Anzahl der Studierenden, die ausführbare Programme erzeugen konnten, und die hohen Erfolgsquoten belegen die Effektivität der Anwendung.

Vor allem der Aspekt „Technik (Verfügbarkeit, Robustheit, Effizienz)“ wurde mit einer durchschnittlichen Note von 3,57 allerdings negativ bewertet. Eine mögliche Erklärung ist, dass die Webanwendung in der damaligen Implementierung unmittelbar nach jeder Änderung des Aktivitätsdiagramms (unter Beibehaltung syntaktischer Korrektheit) Java-Code generierte und diesen, sofern er auch syntaktisch korrekt war, ebenfalls unmittelbar kompilierte und eine entsprechend RoSE-Animation generierte. Die daraus resultierende hohe Anzahl an Server-Anfragen sorgte in Verbindung mit vielen gleichzeitigen Benutzern vor allem am Ende der erlaubten Bearbeitungszeit dafür, dass die Webanwendung zeitweise nicht oder nur eingeschränkt erreichbar war und Studierende die Aufgabe somit nicht bearbeiten konnten. In der Tat wurden andere Aufgaben (außerhalb von WARP), die in der gleichen Woche zu bearbeiten waren, von etwa 290 Studierenden bearbeitet, so dass zu befürchten ist, dass zwischen 15% (Rechteck) und 28% (Labyrinth) der Studierenden WARP nicht verwenden bzw. kein lauffähiges Programm erstellen konnten. Auch die erzwungene Verwendung eines Java-Applets und die damit verbundenen Sicherheitsrisiken könnten ein Grund für schlechte Bewertungen gewesen sein.

Dass die weiteren Aspekte trotz dieser technischen Probleme dennoch als gut (Konzept, Bedienbarkeit, Inhalt) oder befriedigend (Umsetzung, Fairness, Erklärungen) bewertet wurden, werten wir als Erfolg des Ansatzes. Da die Studierenden den Einsatz elektronischer Tools grundsätzlich stark befürworten, wird WARP auch weiterhin eingesetzt, wobei bereits einige Verbesserungen implementiert wurden:

- explizite statt implizite Feedback-Generierung zur Verringerung der Serverlast
- Neuimplementierung des RoSE-Players mit Standard-Webtechnologien
- stochastische Elemente in Aufgaben, um überspezialisierte Lösungen abzulehnen
- interessantere Aufgaben (z. B. durch Verwendung von Mehrspielerszenarien)
- Beseitigung kleinerer Fehler (z. B. unschöne Formatierung des generierten Codes)

Wir erwarten dadurch eine noch bessere Akzeptanz seitens der Studierenden. Vor allem von Mehrspieleszenarien, bei welchen Studierende ihre Roboter in der sogenannten „WARP-Arena“ auch gegeneinander antreten lassen können, erhoffen wir uns eine weitere Erhöhung intrinsischer Motivation.

Literaturverzeichnis

[ASI07] Ali, N; Shukur, Z.; Idris, S.: Assessment system for UML class diagram using notations extraction. IJCSNS International Journal of Computer Science and Network Security. 2007.

[Ba04] Balzert, H.: Lehrbuch der Objektmodellierung: Analyse und Entwurf mit der UML 2. 2004.

[He13] Hermann, F. Erweiterung der Robotersimulationsumgebung RoSE auf Multiagentensysteme mit didaktischem Fokus. Bachelorarbeit, Julius-Maximilians-Universität Würzburg. 2013.

[Hö09] Hörnlein, A.; Ifland, M.; Klügl, P; Puppe, F.: Konzeption und Evaluation eines fallbasierten Trainingssystems im universitätsweiten Einsatz (CaseTrain). In: GMS Med Inform Biom Epidemiol 2009;5(1):Doc07. 2009.

[HQW08] Hoffmann, A.; Quast, A.; Wismüller, R.: Online-Übungssystem für die Programmierausbildung zur Einführung in die Informatik. In DeLFI 2008: Die 6. e-Learning Fachtagung Informatik, S. 173–184, Bonn. 2008.

[If12] Ifland, M.; Hörnlein, A.; Ott, J.; Puppe, F.: Geführtes Tutorsystem mit Unterstützung zunehmender Selbstständigkeit zur Modellierung von UML-Klassendiagrammen. DeLFI 2012 – Die 10. e-Learning Fachtagung Informatik. 2012.

[RJB04] Rumbaugh, J.; Jacobson, I.; Booch, G.: Unified Modeling Language Reference Manual. 2. Auflage. Addison-Wesley Professional. 2004.

[SG11] Striewe, M.; Goedicke, M.: Automated checks on UML diagrams. Proceedings of the 16th annual joint conference on Innovation and technology in computer science education - ITiCSE '11. 2011.