COCLAC - Feedback Generation for Combined UML Class and Activity Diagram Modeling Tasks

Philip-Daniel Beck,[1] Thomas Mahlmeister,[2] Marianus Ifland,[3] Frank Puppe[4]

Abstract: We introduce a newly developed visual programming and UML modeling tool for educational use. It's implemented to provide students with informative feedback during exercises as well as to assess the submissions of students automatically. Tasks combine class and activity modeling aspects as well as some simple programming requirements. The system is web-based and named COCLAC. Visual programming is done through an included UML editor. Created diagrams are syntactically checked and automatically converted into Java code using several conventions. Thereafter, code is executable on a server and semantic correctness can be checked through automatic tests.

Keywords: COCLAC, UML Modeling, Visual Programming, E-Assessment, Feedback Generation

1 Introduction

In this paper we introduce the novel web-based tool *COCLAC* (COmbined CLass and ACtivity tool), a visual programming and UML modeling tool, which has been developed and applied at the University of Würzburg recently. The main motivation for its development was the reduction of manual supervision and correction efforts during UML related exercises of the Software Engineering course which is attended by many students from a lot of different disciplines. This is reached by giving informative feedback during the student's editing process and having automatic assessment of a student's submission. Additionally, the tool is meant to increase the intrinsic motivation of students during doing their exercises by gamificating the task to some extend. Previously developed tools (especially WARP [If14]) at the University of Würzburg already do this for UML activity-diagram based visual programming tasks. However, these tasks only provide modeling tasks for behavior without structure modeling. Nevertheless, success with these tools has inspired the development of the novel tool. Thereby, visual programming is done through class and activity diagram modeling offering new complexity of tasks for students. To do this a web-based UML editor is supplied. Diagrams are syntactically checked and automatically converted into Java code using different conventions. After that the code is executed on a server. Therefore automatic tests can be applied to check semantics of the program. Here we want to share our experiences with the tool and discuss emerged problems.

The paper is structured as follows: The next section gives an overview of related tools. The third part gives a technical overview of the tool as well as a description of the general

[1] University of Würzburg, Informatics VI, Am Hubland, 97074 Würzburg, philip.beck@uni-wuerzburg.de
[2] University of Würzburg, Am Hubland, 97074 Würzburg, thomas.mahlmeister@stud-mail.uni-wuerzburg.de
[3] University of Würzburg, Institute of Computer Science, Am Hubland, 97074 Würzburg, marianus.ifland@uni-wuerzburg.de
[4] University of Würzburg, Informatics VI, Am Hubland, 97074 Würzburg, frank.puppe@uni-wuerzburg.de

work flow that students go through when working on a task. The fourth section gives an overview of usage statistics and difficulties we gathered during the first application of the tool.

2 Related Work

There are several existing tools which are somehow related to our system depending on which aspects you focus on. Some systems deal with the tasks of teaching programming skills or UML diagram modeling, others do code generation from UML diagrams, visual programming or automatic assessment. We introduce several tools which deal with one of these aspects.
Tools for teaching programming skills exist a lot, e.g. Praktomat [KSZ02]. Code may be uploaded to the Praktomat server where it is automatically tested against some pre-defined tests. Therefore only solutions with a certain base functionality can be submitted. COCLAC has a similar work flow by running some tests before submission. Thus, no submission can be made without passing these tests. However, Praktomat is not made for visual programming.
Visual programming is supported by a lot of tools for a bunch of reasons. Some are built just for educational reasons, for example Scratch [Ma10] which is meant to familiarize pupils from age 8 to 16 with general programming skills. Raptor [Ca09] teaches object-oriented programming using UML and flowcharts. It provides syntactic feedback and modeled programs can be executed. Although, it is very similar to COCLAC regarding the general idea, it is not web-based and has no semantic checks for the generated code. Other tools, like FUJABA [NNZ00], are designed for real programming tasks. FUJABA uses UML class and behavior diagrams like collaboration diagrams, activity diagrams and state-charts for visual programming. To make the defined program executable, diagrams are converted into Java code. However, it lacks a feedback component for e-learning purposes.
Other tools that do Java code generation from UML diagrams are UJECOTR, ArgoUML or others. UJECTOR [UNK08] generates code from UML class, sequence and activity diagrams. The generated code is fully functional and comprehensive. Class diagrams are used as frame for the Java class. Methods' code is generated through sequence diagrams which can reference activity diagrams to complete functionality.
[So10] introduce a web-based e-learning tool for UML class diagrams, which focuses on immediate feedback through automatic correction of UML class diagrams, but does not use activity diagrams. GATE [MS13] aims at teaching UML modeling too after being developed for teaching programming skills in Java in the first place. The focus is on UML class and activity diagram modeling. GATE's editor is based on ArgoUML and has been extended to provide special feedback options for UML tasks. However, the analysis is only based on static analysis. Conversion to Java code is not done and tasks are either for class or for activity diagrams. DUESIE [Ho08] is also able to manage tasks for UML diagrams and analyzes these diagrams automatically by comparing it to a minimal sample solution. So none of these tools combines modeling UML structure and behaviour diagrams with providing syntactic and semantic feedback for educational use.

3 COCLAC

3.1 The Typical Workflow

To handle a task the typical workflow is as follows: The student opens the tool via a link from the central Moodle based e-learning platform at the University of Würzburg. There, the description of the task is presented. Next the student creates a class diagram based on the task description. The class diagram needs to contain all attributes and all methods which need to be modeled by activity diagrams in the next step. Furthermore, the class diagram has to be syntactically correct before proceeding with activity diagram modeling. This is checked automatically as soon as the student clicks the corresponding button. If it is not syntactically correct, some errors are shown. The student has to correct the diagram until no errors remain. Afterwards, the student has to visually implement the methods defined in the class diagram by modeling activity diagrams. For each method in the class diagram an activity diagram template is automatically created which needs to be filled with the functionality demanded by the assignment. Thereby, it is possible to model branches and loops through branching nodes or through structured elements using special given activity nodes. Since everything is compiled into Java, normal Java code can be used in certain diagram nodes. The activity diagrams again have to be syntactically correct which is again checked automatically. If there are no errors left, the Java code is generated from both the class and the activity diagrams. This code subsequently needs to be coupled with data and is executed on this data to check semantic correctness. Coupling is done via CSV files. Students may create and upload their own CSV files to test their program on their own data. If the implemented design passes two stages of tests against different data provided by the system, the program may be submitted. After submission, a third check is run against the program. The third data set is not presented to students. Therefore, students can not adapt their programs to the specific test data.

3.2 Technical Overview

The COCLAC web application is built upon a Tomcat server and uses JSP and Java as main back-end technologies. Application data is persisted in a MySQL database. The front-end is built around HTML5 and AngularJS. In the following enumeration we discuss some technical aspects of our system:

1. Authentication is done through Moodle[DT03]. Therefore students do not need to log in to the application with separate credentials. By using links from Moodle courses, necessary authentication data is transmitted automatically.
2. The COCLAC editor (Graditor[Sc14]), which can be seen in Figures 1 (class diagram) and **??** (activity diagram), is the front-end's core component. It is based on HTML5 and provides functionality for creating UML class and activity diagrams.
3. Class diagram syntax checks (meant to check if a diagram can be translated to Java code without problems) are done through a number of defined rules: class names

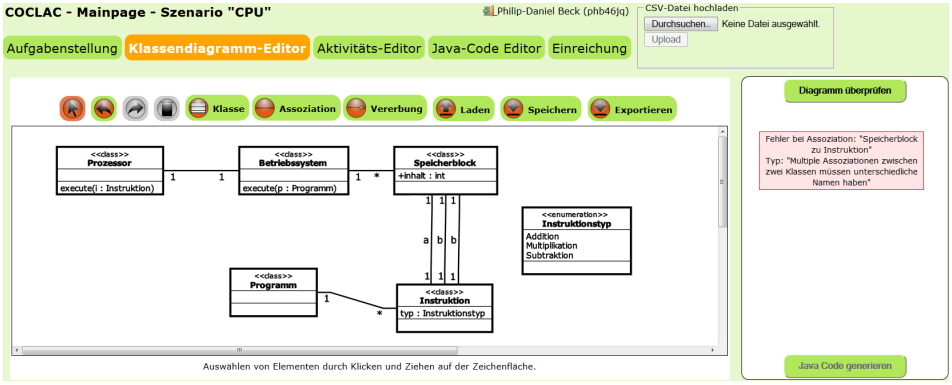


Fig. 1: Web surface of COCLAC showing the class diagram editor with a drawn class diagram. On the right side an error is shown, because two of the associations between *Speicherblock* (Memory Block) and *Instruktion* (Instruction) have the same role name *b*.

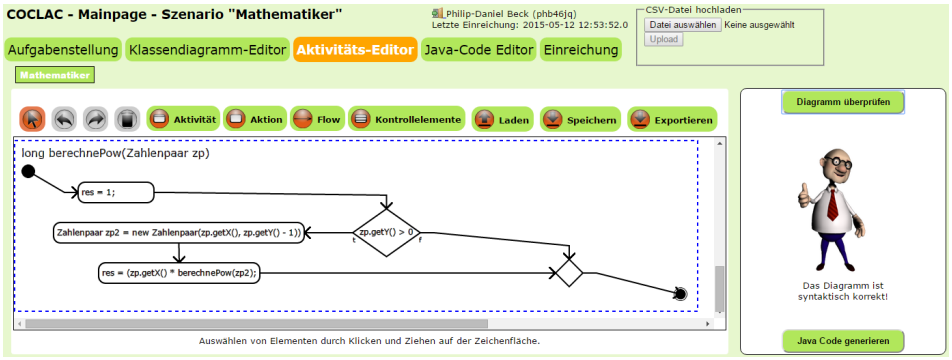
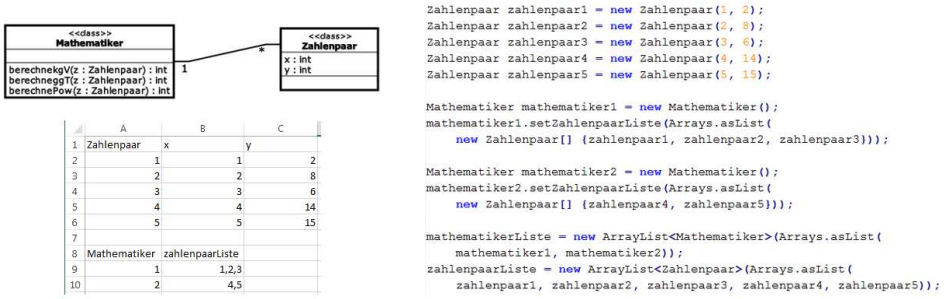


Fig. 2: Depiction shows an activity diagram of the calculation of the pow function in the mathematician task. The variable *res* is defined as the return value of the method by convention.

must be unique; syntax of attributes and methods must be correct; inheritance is checked for absence of circles; if there are several associations between two classes, they need to be distinctly named by roles. If any of these rules is hurt, a related error message is shown in the GUI.

4. The translation from UML class and activity diagrams into Java code is not unique. Therefore, Java code generation is done by sticking to a number of obvious simple rules and defined conventions, e.g. one Java class per UML class, attributes are generated into constructors, getter and setter are generated for attributes.

5. To test the generated code semantically, we implemented possibilities to map data from a CSV file to the programmed model. Figure 3 shows an example. A simple class diagram and an extract from a CSV file. From these two sources the code on the right is generated. However, be aware that this only works if names in the CSV file can exactly be matched to the class diagram model.



	A	B	C
1	Zahlenpaar	x	y
2	1	1	2
3	2	2	8
4	3	3	6
5	4	4	14
6	5	5	15
7			
8	Mathematiker	zahlenpaarListe	
9	1	1,2,3	
10	2	4,5	

Fig. 3: On the left a simple class diagram together with a CSV file containing related test data is shown. The right side depicts a listing of the generated test code.

6. Security policies: As usual for this kind of software, we had to make sure that students are not able to run any bad code on the server. Therefore we used a separate JVM together with the JVM policy file to allow execution of only certain API libraries. To prevent server overload by endless loops, we packed execution of students' code into own processes which run for a maximum of 10 seconds.

7. Feedback is generated at different processing steps. In the first step, feedback is generated for the syntax of a class diagram. As described above, this feedback is generated through rules. The same is done for activity diagrams. The second kind of feedback is generated by running the implemented application. The code is run on input data. The results of this run are compared to the expected results, produced by a deposited sample solution. We want to emphasize that the teaching staff can obtain the output on arbitrary data without having additional effort since it is executed on the sample solution which needs to be provided anyway.

4 Evaluation

In contrast to sole class or sole activity diagram modeling tasks, suitable tasks for the novel tool need to offer structural and behavioral requirements while not being too complex. We had two different tasks that were used during exercise of the software engineering course. The first task *Mathematician* was meant to make students familiar with the tool itself. Therefore the task was not too complex. Students had to model classes for Mathematicians and *NumberPairs*. Each Mathematician keeps a set of number pairs in his mind. On these pairs he can execute different operations, like e.g. greatest common divisor and so on.

The second task was modeling a simple computer consisting of a *Processor*, an infinite number of *Memory Blocks*, an *Operating System* and a couple of *Programs* made of *Instructions* of different types, like *Addition*, *Subtraction* and *Multiplication*. The task was to model a class diagram, which can be seen in Figure 1, as well as activity diagrams for execution of programs. A program is executed by the operating system, which iterates over all instructions of the program. Each instruction is passed to the processor, which executes the instruction considering its type on the associated memory blocks.

—	Math (per User)	Math (all)	CPU (per User)	CPU (all)
User	248	-	128	-
ClassDiagram exists	248	1829	128	1906
ClassDiagram correct syntax	246	1379	117	1003
ClassDiagram correct classes	244	-	103	-
ActivityDiagram exists	237	4457	91	2265
ActivityDiagram correct syntax	228	3313	72	1007
JavaCodeCompile correct	228	1880	65	395
FirstSampleCheck correct	222	-	61	-
SecondSampleCheck correct	212	-	50	-
SubmissionSuccess	210	-	50	-

Tab. 1: The table shows some user statistics for the two tasks. For each tasks the different workflow steps are listed together with both the number of users that succeeded in this step ('*per User*'), and the total number of recognized attempts in this step ('*all*'). E.g., 237 users attempted to create a correct activity diagram for the mathematician task. Therefore, 4457 activity diagrams have been checked for correct syntax. Out of these 3313 had correct syntax, and 228 users reached such correct diagrams.

The tool has been used during the software engineering course at University of Würzburg in the summer term in 2015 and has given us some data and experience for further improvements. The course was visited by about 300 students from different degree courses like computer science, human-computer-systems, business mathematics, business informatics, computational mathematics, spacecraft informatics, and lectureship computer science. Table 1 provides some statistics on the two tasks. The first task was solved more frequently. This has several reasons. Since we could not oversee all complications that might occur during usage of the new tool, we only made the Mathematician task mandatory. Additionally, to further reduce complexity for students we wrote a tutorial for some parts of the Mathematician task, which eased the processing of this task a lot. The CPU task on the other hand was voluntary but after all was giving some bonus points. Nevertheless, it was much more difficult in our opinion. This was confirmed by the evaluated data: the students had problems at different points in the process. Successful editing of the different workflow steps decreased with each step. Table 1 shows that the CPU task was tried by 128 students and 117 achieved to pass on with a syntactically correct class diagram. For some reason we do not know, not everyone who had a correct class diagram also tried to go on with work. Those who tried processing the activity diagram part did mostly succeed: 72 of 91 achieved a correct activity diagram regarding syntax. By comparing the number of all class diagram checks with the number of all correct class diagram checks you can see that 1003 of 1906 class diagrams where syntactically correct. This means by average each student needed about seven attempts to build a syntactically correct diagram. Furthermore, each user needed several iterations to build a class diagram which is also semantically correct. Thus each student has about eight different syntactically correct class diagrams in average. To estimate how much students tried class diagram processing seriously we calculated the number of users with class diagrams containing all necessary classes. This was achieved by 103 of 117 users.

We also evaluated some error statistics for class diagram creation to see the most common problems. An overview is given in Table 2. As you can see, some errors are more frequent than others. A lot of errors only occurred for the second task since the Mathematician task was rather easy.

Error Type	Mathematician	CPU
Illegal use of associations or inheritance with enumerations	0	148
Invalid return value	214	390
Class names should start with a capital letter	7	94
Classes may not have the same name	5	10
Attributes, methods and associations need to start with a letter	168	294
Methods or attributes in the same class may not have the same name	14	1
Cyclic inheritance is not allowed	0	0
A class may only inherit from another class	0	21
Multiple associations between a class need to be named	0	5
Multiple associations between a class need different names	0	2
Class names need to start with a letter	10	22
Use of an invalid data type	31	126
Names of methods and attributes should begin with a small letter	32	48

Tab. 2: Error statistic showing the occurrence of some common errors during class diagram syntax checks.

During the first usage of the tool, we faced some difficulties. The used mapping between data and classes is difficult since it severely restricts design choices: To map data to classes, names in data specification and class diagrams need to be identical. To overcome this ambiguity, we tried to emphasize the expected names by writing ”use the name *xyz* for this variable” in the task description. Nevertheless this made the task less variable and a few correctly modeled assignments couldn’t be tested automatically. Since we did not give sufficient feedback for this problem in the beginning stages (during creation of class diagrams), we did face related problems during submission stage when data was tried to be mapped against Java code.

5 Discussion and Conclusion

As the last difficulty shows, it is difficult to anticipate all kinds of user errors to provide adequate feedback. Therefore an iterative error analysis as done in Tab. 2 and a subsequent update of the error messages for the user is of key importance for an automatic feedback generation tool. Although there is much room for improvements, COCLAC has shown that the general idea to combine the processing of class and activity diagrams in one task works in practice. The modeling tasks convey the connection between structural and behavioral types of diagrams as well as providing an environment for learning concepts of object-oriented and visual programming. Although, the CPU task was voluntary we had several students having submitted the task successfully. This shows generally that students can solve moderately complex tasks with the tool. In future work we need to extend the

tool by adding improved feedback for typical errors which follow from a wrong respective unexpected naming of classes or attributes, as described above. Another area for improvement is automatic assessment for tasks. So far the students get all points if the tests pass and get no points if a test fails. By adding separate evaluation components for class and activity diagrams the tool could give points for every done step. Moreover, since some students in that early part of their studies lack knowledge of Java we could replace the need of Java code during activity diagram modeling through the introduction of new semantic nodes, e.g. for lists or output of variables.

Literaturverzeichnis

[Ca09] Carlisle, Martin C.: Raptor: A Visual Programming Environment for Teaching Object-oriented Programming. J. Comput. Sci. Coll., 24(4):275–281, April 2009.

[DT03] Dougiamas, Martin; Taylor, Peter: Moodle: Using Learning Communities to Create an Open Source Course Management System. In: Proceedings of EdMedia: World Conference on Educational Media and Technology 2003. Association for the Advancement of Computing in Education (AACE), Honolulu, Hawaii, USA, pp. 171–178, 2003.

[Ho08] Hoffmann, Andreas; Bode, Markus; Nichau, Marco; Garbas, Michael; Hellweg, Christoph; Quast, Alexander; Wismüller, Roland: DUESIE - Ein Online-Übungssystem zur Informatik Ausbildung. In: Workshop Proceedings der Tagungen Mensch & Computer 2008, DeLFI 2008 und Cognitive Design 2008. Logos Verlag, Berlin, p. 416, 2008.

[If14] Ifland, Marianus; Herrmann, Felix; Ott, Julian; Puppe, Frank: WARP - ein Trainingssystem für UML-Aktivitätsdiagramme mit mehrschichtigem Feedback. DeLFI 2014 - Die 12. e-Learning Fachtagung Informatik, 2014.

[KSZ02] Krinke, Jens; Störzer, Maximilian; Zeller, Andreas: Web-basierte Programmierpraktika mit Praktomat. Softwaretechnik-Trends, 22(3):51–53, 2002.

[Ma10] Maloney, John; Resnick, Mitchel; Rusk, Natalie; Silverman, Brian; Eastmond, Evelyn: The scratch programming language and environment. ACM Transactions on Computing Education (TOCE), 10(4):16, 2010.

[MS13] Müller, Oliver; Strickroth, Sven: GATE - Ein System zur Verbesserung der Programmierausbildung und zur Unterstützung von Tutoren. In: ABP. 2013.

[NNZ00] Nickel, Ulrich; Niere, Jörg; Zündorf, Albert: The FUJABA environment. In: Proceedings of the 22nd international conference on Software engineering. ACM, pp. 742–745, 2000.

[Sc14] Schneider, Christian: Konzeption und Entwicklung grafischer Editoren für E-Learning-Systeme mit Standard-Webtechnologien. Masterarbeit, Universität Würzburg, Deutschland, 2014.

[So10] Soler, Josep; Boada, I; Prados, F; Poch, J; Fabregat, R: A web-based e-learning tool for UML class diagrams. In: Education Engineering (EDUCON). IEEE, pp. 973–979, 2010.

[UNK08] Usman, Muhammad; Nadeem, Aamer; Kim, Tai-hoon: UJECTOR: A tool for executable code generation from UML models. In: Advanced Software Engineering and Its Applications (ASEA). IEEE, pp. 165–170, 2008.