Teaching UML Skills to Novice Programmers Using a Sample Solution Based Intelligent Tutoring System

Joachim Schramm, Sven Strickroth, Nguyen-Thinh Le and Niels Pinkwart
Clausthal University of Technology, Department of Informatics
{joachim.schramm,sven.strickroth,nguyen thinh.le,niels.pinkwart}@tu clausthal.de

Abstract

Modeling skills are essential during the process of learning programming. ITS systems for modeling are typically hard to build due to the ill definedness of most modeling tasks. This paper presents a system that can teach UML skills to novice programmers. The system is "simple and cheap" in the sense that it only requires an expert solution against which the student solutions are compared, but still flexible enough to accommodate certain degrees of solution flexibility and variability that are characteristic of modeling tasks. An empirical evaluation via a controlled lab study showed that the system worked fine and, while not leading to significant learning gains as compared to a control condition, still revealed some promising results.

Introduction

The number of students in Informatics and Computer Science is decreasing across many universities (Matthíasdóttir, 2006). This tendency might be explained by the fact that courses on programming, which constitute an indispensable part of studies related to Informatics and Computer Science, are considered difficult by many students. McCracken and colleagues showed that many students still have a lack of programming competence even after a full year of programming instruction (McCracken et al., 2001). Why is programming so difficult subject for novice students? The answers to this question are diverse. One aspect is that programming requires a combination of surface and deep learning. Deep learning here means that students have to concentrate on gaining an understanding of a topic while surface learning means memorizing facts. Thus, programming cannot be learned solely from books: Instead, students have to learn programming by developing algorithms and systems by themselves to deepen their understanding (Lahtinen et al., 2005).

Computational modeling is one of the essential steps during the process of developing a computer program (Feddon & Charness, 1999). However, the first step most novice programmers carry out when they start to solve a programming problem is typing in code - often, students tend to analyze a task and their system design in the middle of the coding process (Pintrich et al., 1987; Perkins et al., 1989, p. 257). They could clearly benefit from more modeling skills (and from better meta-cognitive skills, but this is not subject of this paper). As such, systems capable of helping students acquire such modeling skills would be beneficial. Indeed, a few of these systems have been proposed. They all face the issue that most modeling tasks that go beyond the essential basics are ill-defined (Lynch et al., 2009) and that, consequently, computer based support (particularly, ITS support) is hard to implement and costly. In this paper, we propose a system that teaches UML skills to novice programmers. The system is "simple" in the sense that it only requires an expert solution against which the student solutions are compared, but still flexible enough to accommodate certain degrees of solution flexibility that are characteristic of modeling tasks.

ITS Tools for Teaching Modeling - A Survey

While not as many as in the domain of programming, there have been few attempts for building ITSs for computational modeling. Applying the classical Constraint Based Modeling approach, Baghaei and Mitrovic (2007) developed Collect-UML, a collaborative constraint-based tutor for UML class diagrams. This system supports students in problem-solving both individually and collaboratively. First, students create UML class diagrams individually using the system's feedback. Then, they join into small groups to create group solutions. In the collaboration mode, the system compares the group solution to individual solutions of all group members. It provides feedback to the group solution, and at the same time it provides feedback on collaboration. An evaluation showed that students using this tool acquired more knowledge on effective collaboration than a control group by 1.3 standard deviations. Yet, using this system, students

are restricted to only using noun/verb phrases provided in the problem description as labels for classes, attributes, and methods.

Soler et al. (2010) developed another tool for teaching UML class diagrams. This tool is able to automatically assess UML class diagrams provided by a student. For each problem, the system has a set of correct solutions. When a solution is entered by the student, the system compares it against the specified correct solutions. The system selects the correct solution which is most similar to the one proposed by the student and returns a feedback message to the student. Using this system, students are restricted to use noun phrases indicated in the problem description to specify attributes for a class. There is no restriction on names of classes or relationships. Classes are assessed by considering the set of attributes attached to them. Relationships are evaluated in terms of the classes they relate to. It has been reported that students who used this tool got better examination marks than students who did not do so.

Similarly, several other tutoring systems for UML diagrams check a student's diagram based on a single expert's solution. This includes CIMEL, a system that supports students learning object-oriented analysis and design as problem-solving skills (Blank et al., 2005). Also, a system proposed by Ali et al. (2007) checks student's class diagrams by identifying the differences between the student's class diagram and an expert's class diagram. Evaluation results of these systems have not been reported.

All the reviewed tutoring systems are intended to support students in their skills of modeling and creating UML diagrams. They share the limitations that only class diagrams are supported (and, thus, only static code structures but not algorithmic processes can be taught) and that solutions are often only checked against a single pre-defined "expert solution" (or a small set thereof) and that alternative correct solutions which differ from the expert solutions are not accepted.

The system presented in this paper attempts to overcome these limitations. It supports the design process with UML class and activity diagrams, and it accepts variations of good solutions.

System description

Architecture

Several courses at Clausthal University employ a web-based exercise management system named GATE (Strickroth, 2009). The GATE system is based on a 4-layer (GUI, application, persistence, data) client/server architecture. The exercise data is stored on the server in a central place in order to administer it more easily. Students can access the GATE system using a web-browser from anywhere.

The MFS system, presented in this paper, consists of two components: an adapted version of the open source modeling tool ArgoUML[1], which can be started directly from the GATE system in a browser using Java Web Start technology, and some add-ons to GATE itself. Solutions developed in the modified ArgoUML tool can be uploaded to the GATE system, where feedback is calculated and sent back to the modeling tool.

Teachers can use ArgoUML to model and save either class diagrams or activity diagrams as sample solutions. These sample solutions can be associated to task descriptions. In addition, teachers can create a UML constraint test for each modeling task and they can specify the maximum number of feedbacks that a student is allowed to ask for (in order to prevent try & error strategies).

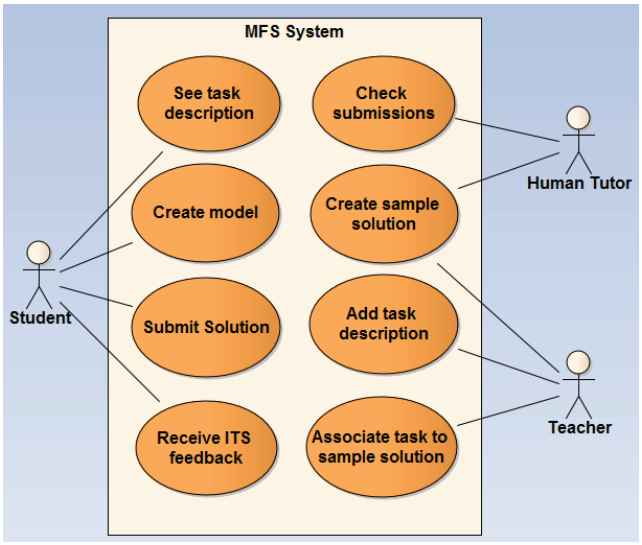

Figure 1. Use cases of the MFS system

Solution Analysis and Feedback Provision

Students use the adapted ArgoUML for solving their tasks. Both class diagrams (for static code structures) and activity diagram (for algorithms) are supported. The diagram type is automatically detected based on the task description in GATE; it is then preselected in ArgoUML. Furthermore, the task description is embedded into the MFS-modified ArgoUML, so that it is visible for the students during the whole modeling process.

Over a specific function (called "Export2Gate"), students can send their task solutions to the server without the need of saving it and manually uploading it to the GATE submission system. If a student wants to edit an already uploaded submission, ArgoUML automatically loads the previous submission upon start. After a submission using the "Export2Gate"-function, a further window containing a "feedback button" is shown.

[1] http://argouml.tigris.org/

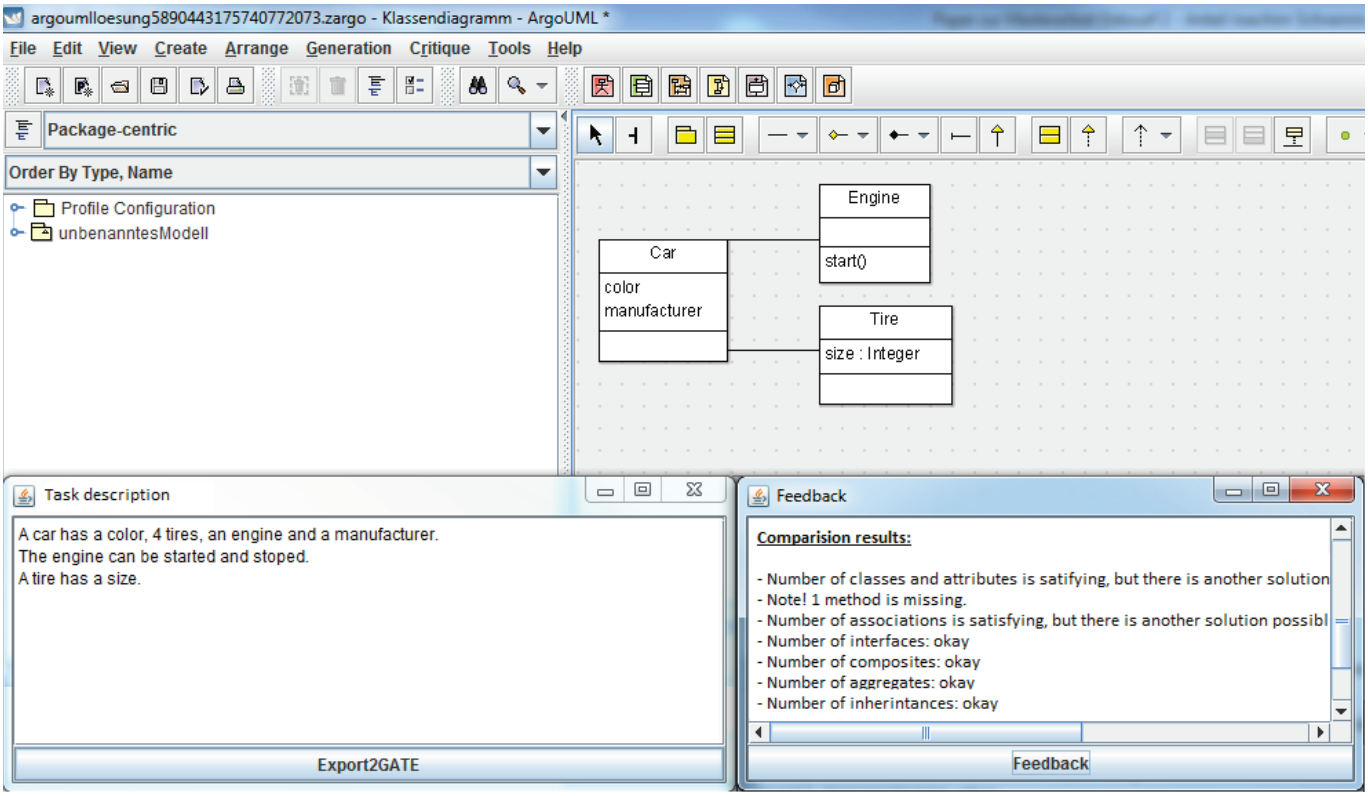

Figure 2. Screenshot of the MFS with feedback

If a student presses the feedback button, the UML constraint test finds out whether the sample solution is an activity diagram or a class diagram, and stores information about the sample solution and the student solution. For class diagrams, this information comprises:

- Number and names of classes
- Number and names of attributes
- Number and names of methods
- Number and names of associations
- Number and names of interfaces
- Names of the interfaces
- Number of composites
- Number of aggregates
- Number of inheritance
- Number of interface implementations
- Association to classes allocation
- Attributes to class allocation
- Methods to class allocation
- Methods to interface allocation

For activity diagrams, the stored information contains:

- Number of start states
- Number of end states
- Number of forks
- Number of joins
- Number of junctions
- Number of signal events
- Number of action states
- Number of transitions
- Number of loops

Once a student asks for feedback, the elements of the sample solution are compared against the elements of the student solution. Most of these comparisons are based on a numerical comparison of the elements (see also Soler et al., 2010). Feedback of the following type is then provided:

- Note! <number of missing element(s)> <name of element> missing.
- There is/are <number of elements> <name of element>, this is too much.
- Number of <name of element> okay.

In addition, the names of the elements are compared to the sample solution and corresponding feedback is given to the student. Methods and attributes are examined for correct allocation to classes or interfaces. Finally, the allocations of associations are examined.

In order to increase the freedom required for modeling tasks, the number of attributes and the number of classes are treated differently (to take into account that often, an

aspect can be modeled with a class or with an attribute). The number of attributes plus the number of classes in the sample solution is compared against this summation in the student solution. If the number of classes plus the number of attributes in the sample solution is larger than this sum in the student solution, the student receives the following feedback: "Note! *<number of missing element(s)>* class(es) or attribute(s) missing."

If the number of classes plus the number of attributes in the sample solution is smaller than this sum in the student solution, the student receives the following feedback: "There is/are *<number of elements>* class(es) or attribute(s), this is too much.".

If the number of classes plus the number of attributes in the sample solution is equal to this summation in the student solution, and if the number of classes in the sample solution is equal to the number of classes in the student solution (this implies that the numbers of attributes are equal too), the student receives the following feedback: "Number of classes and attributes okay."

If, however, the number of classes plus the number of attributes in the sample solution is equal to this summation in the student solution, but the number of classes in the sample solution is unequal to the number of classes in the student solution, then the student receives the following feedback: "Number of classes and attributes is satisfying but there is another solution that is possibly better."

A similar calculation process is conducted to determine feedback for student-created activity diagrams, based on the indicators listed above. In particular, the number of loops, conditions, junctions and forks/joins are taken into account while calculating the differences between sample solution and student solution.

Study description

Hypotheses and Design

We sought to evaluate the MFS system with respect to the hypothesis that the use of this tool helps learners acquire modeling skills better than a normal UML tool (without feedback features) can do. In addition, we were interested in comparing MFS to a typical University exercise group situation where one human tutor is available for a group of students.

The MFS was developed for novice programmers. Because of this, we sought study participants with no or only very limited programming and modeling knowledge. Altogether, 30 persons participated in our study. These were assigned randomly to one of three conditions (A, B, C). In condition A, students worked with the aid of the MFS. The maximum number of possible feedbacks was limited for each task based on the degree of task difficulty. In condition B, the students were supposed to solve the tasks without any aid – here, the students used the standard ArgoUML for modeling. In condition C, the solution of the tasks was done with ArgoUML and the support of a human tutor. Here, the human tutor was instructed to give the same maximum number of feedbacks that was available in condition A (e.g., answer only to 5 questions of a student if in condition A, the students can ask for ITS feedback 5 times). The human tutor was also asked to give roughly the same kinds of feedback that the ITS would have given but were free in their formulation (e.g., if the ITS had given the hint that three methods were missing, the human tutor would not have gone beyond this level of specificity, but was allowed to use different formulations, gestures, etc.).

The participants had to fill out two questionnaires in the study. A pre-questionnaire contained demographic questions, and questions about programming and modeling skills. A post-questionnaire contained usability questions, questions about the automatic feedback, and questions about self-evaluation.

After the pre-questionnaire, an introduction to UML diagrams was given. Further, the modeling tool ArgoUML and/or the MFS system was presented and explained. Then, each group was told how they would have to solve the tasks. Students in all conditions had to solve the same simple UML tasks (3 class and 3 activity diagrams) in the same time (70 minutes).

Results

To measure overall system usability, the widely accepted System Usability Scale (a questionnaire of ten items) was employed. With a total result of 80.25, the MFS scored very high (Tullis et al. 2008) – as such, no usability confounds to the study results can be assumed.

For the data analysis, three students were excluded who did, apparently, not have the language skills required to understand to task. The remaining analysis is thus based on 27 subjects (10 in condition A, 9 in condition B, 8 in condition C). An ANOVA yielded that the total time on task did not differ between conditions. The participants in condition A needed, on average, five minutes longer than their counterparts (Fig. 3) – yet, this difference was not statistically significant ($F(2, 24)=1.93$, $p>.1$).

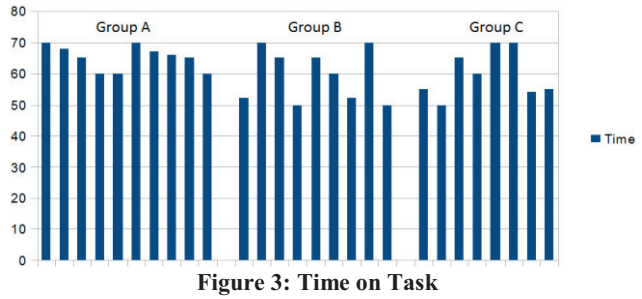

Figure 3: Time on Task

After the study, all solutions were manually assessed by a human UML expert. Concerning the solution quality, there was a significant difference between conditions. The total maximum score a student could get was 104. The group averages were 94.4 (A), 82,2 (B) and 100,9 (C). An

ANOVA showed that this difference was significant ($F(2,24)=5.8$, $p<.01$). A Tukey HSD Test showed that the significant difference was between the human tutor condition (C) and the plain UML tool condition (B).

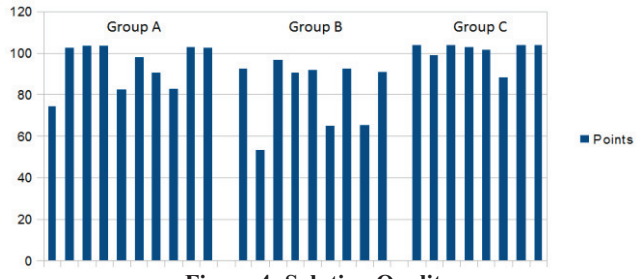

Figure 4: Solution Quality

Some of the results of the post-questionnaire are interesting. In the post-questionnaire, students of condition A stated that they liked the feedback given by the MFS tool (answer mean: 4.3 on a 0-5 scale). This was even higher than the mean answer (4.0) to the corresponding question for condition C about the feedback of the human tutor (answer mean: 4.1) – even though this difference was not statistically significant ($p=.68$). Also, when asked if they agree to the statement that they learned a lot during the study, students in condition A confirmed this statement (4.4 on a 0-5 scale) stronger than the students in the other conditions (3.3 for condition B, 3.8 for condition C) did. An ANOVA showed that this difference was not statistically significant, though ($p=.08$).

A manual inspection of all the models created in condition A showed that in no instance, misleading feedback (i.e., critique of a correct solution, or no critique of a poor solution) was given. This was true for both the class diagrams and the activity diagrams. We also collected evidence as to how the students used the ITS feedback. Overall, the use of the feedback messages was as intended by the MFS designers and lead to an improvement of the models. Examples: In task 1, many subjects chose wrong UML association types and corrected this after the IST feedback (in all but one case). In some tasks, students tended to create class attributes instead of methods (e.g., for "turn engine on" or "turn engine off" in task 1). The ITS feedback was helpful for students to detect and change this. In the activity diagrams, lacking loops were detected by the ITS, and most students were able to add correct loops after the corresponding feedback.

Discussion

At first sight and from a quantitative statistical perspective, the study did not yield the desired result. The main hypothesis that the MFS tool supports learning UML skills better than a standard UML tool can do was not confirmed at the .05 significance level. Yet, there are still some positive and promising results. First, the results of the study indicate that the lack of statistical significance might be due to the relatively low number of subjects. Indeed, the average score of condition A was much higher than the score of condition B (Cohen's $d = 0.76$). Additionally, the fact that condition C did best in the study (although not significantly better than the MFS condition) was not really a surprise: human tutors still do better than ITS systems in most cases, and the tutor-student relationship of 1: 8 in our study was very comfortable for the students (yet, not realistically feasible at public universities in Germany). Another positive aspect of the study is related to a possible field use of the system in real classroom settings: the students expressed that they liked to work with the ITS system, and there are also clear practical benefits at the learning management level (e.g., the integration with the GATE system facilitates the workflow).

Also aside from the empirical insights, there are some contributions of the MFS system to the ITS field: the analysis and feedback methods in the system allow for a simple and low cost development of tutors, but still the mechanisms are flexible and accommodate multiple solution variations – a challenge that any ITS for modeling has to face. In that context, it is remarkable that the "shallow" criteria that the system uses to analyze solutions have, apparently, worked and did not lead to misleading feedback. One can of course critically remark that the MFS system still restricts the flexibility of learners (since only one solution and variants are accepted). Yet, such an approach may be acceptable for novices that work on beginner's tasks, while it is probably less appropriate for more advanced, complex modeling tasks that come with a greater variation of possible solutions.

Conclusion

This paper presented MFS, an ITS for modeling that consists of a UML modeling tool and a web-based exercise management system. MFS is (to our knowledge) the first UML tutor that supports not only class diagrams but also activity diagrams, and it does so by comparing the student solution to an example solution, tolerating certain degrees of solution variations that are typical for modeling tasks. The design of MFS allows for a simple (and thus cheap) authoring of tasks, at the expense of a theoretical possibility of giving inappropriate feedback since not all good solutions might be recognized as such and since the "shallow" check of solutions could be prone to overseeing errors. Yet, in the context of training novices, these potential issues did not come up in practice: the system worked fine and, while not leading to a significant learning gain as compared to a control condition, still gave reasons to continue with this line of research and development.

In future work, we will extend the MFS system to other UML diagram types, and will further fine-tune the system feedback. Also, we are planning a field test of the system in a larger University class with approx. 300 students

(trying to find out to what extent the ITS use help reduce human tutor time), and are planning to devise methods that allow for incorporating multiple sample solutions (and variations thereof) into MFS.

References

Ali, N.H. et al. (2007). *Assessment system for UML class diagram using notations extraction.* International Journal of Computer Science and Network Security, Vol. 7, No. 8, pp. 181-187.

Baghaei, N. & Mitrovic, A. (2007). *Evaluating a collaborative constraint based tutor for UML class diagrams*. Proceedings of the 13th International Conference on Artificial Intelligence in Education, pp. 533-535.

Blank, G. et al. (2005). *A web based ITS for OO design*. Proceedings of The 12th International Conference on Artificial Intelligence in Education.

Feddon, J. & Charness, N. (1999). *Component relationships depend on skill in programming*. Proceedings of 11th Annual PPIG Workshop, University of Leeds, UK.

Lahtinen,. E. et al., 2005. *A study of the difficulties of novice programmers.* Proceedings of the 10th annual SIGCSE Conference on Innovation and Technology in Computer Science Education, pp.11-18.

Lynch, C., Ashley, K. D., Aleven, V., & Pinkwart, N. (2009). *Concepts, Structures, and Goals: Redefining Ill Definedness*. International Journal of Artificial Intelligence in Education, 19(3), 253-266.

Matthíasdóttir, A. (2006). *How to teach programming languages to novice students? Lecturing or not?* Proceedings of the International Conference on Computer Systems and Technologies, pp. 1-6.

McCracken, M. et al. (2001). *A multi national, multiinstitutional study of assessment of programming skills of first year CS students.* In Working group reports from ITiCSE on innovation and technology in computer science education, New York, NY, USA, pp. 125–180. ACM.

Pintrich, P. R. et al. (1987). *Students' programming behavior in a Pascal course*. Journal of Research in Science Teaching 24 (5), 451–466.

Perkins, D. N. et al. (1989). *Conditions of learning in novice programmers*. In E. Soloway & J. C. Spohrer (Eds.), Studying the Novice Programmer. Lawrence Erlbaum Associates.

Soler, J. et al. (2010). *A web based e learning tool for UML class diagrams*. Proceedings of IEEE Education Engineering (EDUCON), pp. 973 – 979.

Strickroth, S. (2009). U*nterstützungsverfahren für Tutoren bei der Online Abgabe von Übungsaufgaben*. Bachelor Thesis. Clausthal University of Technology, Department of Informatics

Tullis, T., Albert, B., 2008: *Measuring the User Experience, Collecting, Analyzing and Presenting Usability Metrics*, Morgan Kaufmann.

Annex: Task descriptions

Task 1: Create a UML class diagram for the following description: A car is specified through its producer, its name and its color. Tires and the motor are part of a car. A motor is specified by output power and cylinder capacity. A motor can be started and stopped. A tire has a certain pressure. A person has a surname and a last name, persons can buy cars. Additionally, a person can have a bank account, which is specified by an account number.

Task 2: Create a UML class diagram for the following description: A soccer player has a name, a date of birth and a number. A soccer player is part of a team, and a team cannot exist without players. A team is characterized by its name, points, scored goals and goal difference. There are four player types: a player can either be goalkeeper, defender, midfield player, or striker.

Task 3: Create a UML class diagram for the following description: Computers contain microprocessors and heat sensors. Computers have a specific performance and can be started and stopped. Microprocessors have a specific size, and heat sensors implement the technical interface "IFSensor". This interface allows for activation and reading.

Task 4: Create a UML activity diagram for the following description where Tim wants to cook Spaghettis. He first boils water and then adds the Spaghettis to the boiling water. The Spaghettis then remain in the pot for 10 minutes.

Task 5: Create a UML activity diagram for the following description where Tim thinks about what happens after the assessment of written exam exercises. First, the points are summed up. If the student has less than 100 points, he failed and nothing else happens. If he has at least 100 points, he passed and his mark is calculated. Then, the mark is published and, in parallel, sent to the University administration.

Task 6: Create a UML activity diagram for the following description where Tim comes from classes and wants to watch soccer – but finds out that there is no cold beer in the fridge. First, he puts the beer in the fridge. Then, he waits for 10 minutes and tests if the beer is cold enough. If the beer temperature is 7 degrees Celsius or lower, he takes it and drinks it. Otherwise, he waits another 10 minutes and tests again until the beer is cold enough.