Debugging Students' Debugging Process

Axel Böttcher, Veronika Thurner, Kathrin Schlierkamp, Daniela Zehetmeier
Munich University of Applied Sciences
Faculty of Computer Science and Mathematics, Lothstr. 64, D-80335 München
E-Mail: {axel.boettcher,veronika.thurner,kathrin.schlierkamp,daniela.zehetmeier}@hm.edu

***Abstract*—In this paper we present accumulated results from two years of experience with a teaching unit on debugging Java programs. With this special teaching unit, we strive to foster the debugging skills of our students. Students were asked to find different defects in given code, to analyze these and finally to fix them. As well, students were requested to document their approach in writing. The achieved results ranged from "all bugs found and fixed" to "completely lost in the code". When analyzing these results, we discovered that the debugging skills of our students seem to correlate with some non-technical skills that are essential base competencies in software engineering, such as the ability to work in a systematic way. This implies that for improving our students' debugging skills, it is helpful to address not only the technical aspects of debugging, but to foster the required base competencies as well.**

Keywords — Computer science education; computing education; Software engineering; Software debugging; educational methods; soft skills and competencies; Just-in-Time Teaching

I. Motivation

The ability to troubleshoot, i.e. to detect deficiencies in technical systems and to solve the underlying problems, is a skill that every engineer must have, no matter whether they focus on hardware or software based systems. Even more, engineers should be able to troubleshoot efficiently and thus systematically, since an unsystematic approach would rise costs without producing new features. In this paper, we concentrate on software engineering education and thus software based systems, but our findings may very well be transferred to other engineering disciplines.

In software engineering, the ability to detect and correct deficiencies in software systems comprises, among others, the skill to *debug* existing code. Debugging is a very complex skill, which incorporates several other competencies, such as abstract and logical thinking, working systematically or working in an accurate manner. Moreover, when professional software engineers debug a software system, they make use of a host of strategies that they implicitely derived from years of experience. Quite often, for debugging experts these strategies manifest themselves as a gut feeling rather than a formal sequence of explicit arguments.

From our teaching experience, the process of debugging is very difficult to learn for novice programmers. Considering the complexity of the task and the obvious lack of experience in programming novices, this is hardly surprising. Moreover, although most IDEs support the debugging process by powerful tooling (the *debugger*), students usually do not use this tool support of their own accord. Instead, novice students often apply a mixture of more or less random visual inspection, coupled with print-statements to visualize process flow and the development of variable values.

Despite the fact that debugging is a highly essential skill for every professional software engineer, and learning this skill is obviously a difficult undertaking, literature does not yet define any established best practices for *teaching debugging*. Rather, in teaching practice, when students learn how to code, they usually need (and are expected to develop) the skill of debugging as well. Thus, in many introductory programming classes, both skills are developed to a certain extent in parallel, with a strong explicit focus on coding skills – and debugging as a necessary, but sometimes neglected, side effect. Only few authors [2] claim to explicitely separate debugging skills from coding skills in their teaching, and to design special lectures for either of these topics.

To remedy this situation, we attempt to equip our students with explicit strategies for finding bugs. As we address students in their freshmen year, these strategies must be simple enough to be understandable for programming novices. At the same time, they have to be effective enough to let students experience the value of a systematic debugging process, as compared to their approach of random visual diagnosis. In addition to learning initial debugging strategies, we want our students to master the efficient usage of the IDEs debugger as a matter of course.

Note that throughout this paper, we use the term bug synonymously with defect. With respect to the IEEE Standard Classification for Software Anomalies [1], *defect* is the correct term for an imperfection or deficiency in software where it does not meet its requirements or specifications.

II. Goals

All in all, we strive to improve our students' debugging skills. To achieve this, we attempt to design an approach for teaching systematic debugging in an effecive way. This includes both the application of a systematic debugging process and the usage of the IDE's debugging tool support. As well, we want to induce in our students an awareness that the skill of debugging is highly relevant for their later professional career as software engineers, and thus motivate our students to acquire this skill.

Since debugging is a highly complex skill, we assume that several non-technical base competencies are helpful (or even necessary) for acquiring it, such as the ability to work systematically. If this hypothesis holds true, it would be necessary to

develop these base competencies to a certain extent in our students, to enable them to develop debugging skills in the first place. Therefore, we want to investigate into correlations of selected base competencies and debugging skills.

III. Related Work

As theoretical background for our work, we have to consider literature on debugging itself, work related to teaching debugging and publications on the required base competencies.

A. Debugging

Almost all deficiencies in complex systems are still detected individually [2]. People do make mistakes and cannot always sufficiently grasp complex systems.

Spinellis [3] discusses bug detection strategies and mentions that the work we invest in a debugging session has only ephemeral benefits. As a consequence, from a business point of view, the process of debugging is not considered to be sustainable, even though it might be very expensive. Therefore, all mechanisms for preventing bugs or detecting bugs early on should be employed, such as proper software design and unit testing right from the beginning of writing code. From a business point of view, this implies the necessity for systematic and goal-oriented approaches to find and correct defects in software.

The process of debugging has been analysed in numerous investigations. The approaches range from a technical point of view (e.g. [4]) to psychological aspects as in the study by Gilmore [5]. An easily understandable, vivid description of a defect location process is given by Gauss, namely the "Wolf Fence Algorithm" [6] as a metaphor for the important principle of *Divide and Conquer* applied in debugging.

B. Teaching Debugging

McCauley at al. [7] provided a comprehensive overview of literature related to the learning and teaching of debugging up to 2008, stating that debugging is under-represented in introductory programming textbooks. They concluded that debugging skills can and should be taught together with program comprehension skills in an integrated way. A couple of years later, Yen [9] et al. came to a similar conclusion. However, until now, the fact that debugging is under-represented in introductory programming textbooks still holds true – with the exception of Barnes and Kölling's book "Objects first with Java" [8].

According to [10], "Hypothesizing about the cause of bugs is an underdeveloped skill". The authors also observe a lack of web-materials that address this topic at an appropriate beginner level.

The current draft of Computer Engineering Curricula 2016 (Curriculum Guidelines for Undergraduate Degree Programs in Computer Engineering [11]) lists the learning outcome "Effectively use a debugger to trace code execution and identify defects in code" in the Software Design core knowledge area, thus emphasizing the relevance of this skill for software engineers to-be.

C. Base Competencies Involved in Debugging

Nagvajara et.al. [14] state that there is a relation between debugging and non-technical competencies such as creative thinking, creative problem solving, and creativity in design. More precisely, these non-technical competencies can be considered as being *prerequisites* for debugging, i.e. they must be developed in a student to a certain extent before this student is able to acquire the skill of debugging in the first place.

McCauley et al. ([7, p. 88]) suggest that "further investigations into the links between programming and debugging ability [...], could provide critical benchmarks for assessing the effectiveness of new debugging pedagogies or interventions". Following [13], links between these two skill areas could be a selection of base competencies that is equally relevant for both skill areas, including e.g. the abilities of *working systematically* and *working accurately* as well as *thinking concretely* or *thinking in an abstract way*.

Among others, Kessler and Anderson [12] address the "ability to trace code" and "the ability to locate errors by parsing the code and matching it with the results obtained". Finally, Katz and Anderson [4] put debugging into an everyday context.

IV. Our Teaching Approach

To foster the debugging skills of our students, we dedicate a teaching unit especially to this topic. This teaching unit comprises several parts, namely a preparatory Just-in-Time Teaching (JiTT) [15] unit, followed by a classroom lecture and finally a lab session exercise. Overall, this teaching unit addresses the top four levels of learning objectives (levels "remember" to "analyse") on debugging that are specified in Table I, according to the revised Bloom-taxonomy for educational objectives [16]. Note that as our class addresses programming novices, we consider the two highest levels "evaluate" and "create" as mainly being out of scope for this specific target group.

From our perspective, every software engineer should learn and adopt the following four aspects of debugging and their relating skills:

1) A software engineer must *be aware* that bug detection can be done in a systematic way [2].
2) A software engineer needs strategies for systematically *finding* defects and their causes, and to gain experience in this area.
3) A software engineer needs bug *fixing* strategies.
4) A software engineer has to *ensure* that if in the future, this bug is accidentally re-introduced, it will be immediately detected.

From our experience, the skills in items three and four are comparatively easy to teach. Once a bug is identified and its cause properly understood, fixing it basically employs the same skills as programming in itself. Therefore, if we assume that we are able to develop programming (i.e. coding) skills in our students, item three is sufficiently dealt with.

Item four requires that once a bug has been identified, we expand our unit tests accordingly, so that they explicitely

TABLE I
LEARNING OBJECTIVES ACCORDING TO LEVELS OF REVISED BLOOM TAXONOMY.

Level	Teaching Goal: Students ...
remember	... define the term debugging. ... name different approaches to debugging. ... state functions of a debugging tool.
understand	... explain in their own words why use of a debugger is reasonable.
apply	... schematically use a debugger in order to visualize program state and program behavior. ... relate debugger's output to their own mental expectation during a debugging session – and they detect a mismatch between expectation and actual state.
analyse	... draw conclusions from observed difference between expectation and actual program state. Thereby they search the reason of the mismatch and form hypotheses for reason and develop a solution.
evaluate	...judge if a given or applied or observed debugging strategy is effective and efficient.
create	... develop new strategies to search and localise software bugs.

cover and validate the corresponding functionality. That way, we can automatically detect this bug if it should happen to reoccur in the future. How the topic of testing can be properly integrated into introductory programming classes has already been discussed elswhere [17].

In contrast to this, the first two items are much more difficult to deal with. Item one involves a change in the students' beliefs, whereas the teaching challenge of item two arises from the fact that debugging strategies consist mainly of implicit knowledge, and thus are rather hard to convey. In the following, we focus on these two items.

Note that the teaching approach presented here evolved over two years of experience, gained from subsequent courses on software development in the bachelor degree programs of computer science, scientific computing and geotelematics. Throughout this process, we used our lessons learned from early runs to continuously improve our approach, especially the lab session assignment.

A. JiTT and Lecture Design

The teaching unit starts with the self-learning-phase of JiTT. Students are requested to read a text on debugging. Afterwards, they have to answer some questions on the content of this text, via the E-Learning platform Moodle. After reading the assignment, if a student should happen to have any questions on its content, they can send these individual questions to the lecturer, again using the E-Learning platform.

To provide a notion of *strategy* and debugging process, in our reading assignment and subsequent lecture we employed the "Wolf Fence Algorithm" [6] as a metaphor for a systematic and efficient approach to debugging. Fig. 1 illustrates our notion of applying this algorithm within a debugging process. As in any other Divide&Conquer-based approach the algorithm always works correctly, regardless of the division line's initial position – however, it might become inefficient.

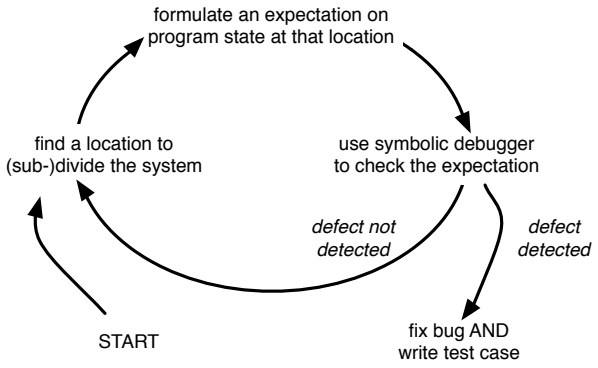

Fig. 1. Debugging process built on the "Wolf Fence Algorithm".

Subsequently to the self-learning-phase of JiTT, the corresponding lecture was used to address the raised questions, as well as any problems that occured. Major parts of this lecture were designed as a life programming session, where the lecturer made his debugging process explicit, while at the same time demonstrating how to use the debugger on a running program.

Note that in the first run of our teaching concept, we observed that students were overwhelmed by the complexity of both the debugger and the debugging process itself. As a consequence, in our second iteration we enhanced our JiTT-material and the corresponding lecture by adding a guideline for the debugging process as well as the usage of the debugger.

B. Lab-exercise Design

After the lecture, to motivate students to discover and actively employ the debugger during lab sessions, we give them a specific assignment that *requires* the use of the debugger. This lab-exercise is based on a programming exercise from a previous semester, done with a different cohort.

Exercise materials comprise the following:

1) A description of the programming task, as it was used in the previous semester with a different cohort.
2) Student-written code that was turned in as solution during the previous semester, including unit tests. We "enhanced" this solution by purposely introducing several errors into this code, which we had collected and identified as "typical" during the previous semester. Furthermore, we cleaned this code up with respect to coding guidelines and Javadocs.
   Note that in our first approach, the test setup in one of the provided test cases was intentionally erroneous. As this proved to be too complex for *all* students in our first attempt, we omitted this kind of bug in our second attempt.
3) A description of the debugging-task itself. Students were asked to identify, analyze and fix any bugs within the provided source code, by applying the Wolfe Fence Algorithm and using the debugger. Of course, students had to hand in their corrected source code. In addition,

TABLE II
OFF-BY-ONE-ERROR.

Off-by-one-error	

Code	`charactaristic = ingredient.getCharacteristic();`
	`for (int k = 1; k <= charactaristic.length() - 1; k++) {`
	`  ...`
	`}`

Error Class	MISCONCEPTION
Description	In the character-by-character processing of a String, the index starts at one, thus skipping the Strings first character.
Solution	Counting starts at zero, not at one.

to make their debugging process explicit, students had to document the process they applied in writing, supported by suitable screenshots from the debugger, visualizing the usage of breakpoints or the variables view. This is a somewhat modfied version of "think-aloud" experiments [9].

4) To help students understand what is expected of them, we provide a sample solution for one selected bug, which comprises the textual elaboration of the debugging process and its corresponding screenshots of the debugger.

The underlying programming exercise focuses on inheritance, classes and methods as well as on algorithms. In terms of content, it deals with elixirs (magic potions) consisting of ingredients of different types (herb, mineral or animal). Each ingredient contains a caloric information and a defined characteristic that is represented by a String with a length of 20, consisting only of X's and O's. An elixir consists of several ingredients combining their characteristic Strings together via an XOR operation.

An elixir has different properties that are determined by the structure of the combined characteristic Strings. For example, the elixier is "poisonous" if its combined characteristic String is a palindrome; or "forces truth telling" (is a veritaserum) if the String contains an even number of X's, and so on.

Initially, the provided source code comprises five failing unit tests. These failures are caused by three defects in the code, as described in Tables II, III and IV. The classification of these defects follows [18].

V. ANALYSIS APPROACH

To both evaluate our approach and better understand our students' debugging process, we thoroughly analysed the results that our students handed in. Note that in this analysis, we focus on the results of winter term 2015/16, and thus the most mature version of our teaching approach. Overall, 39 textual descriptions were handed in. Nine of these were created by teams of two. Hence, a total of 48 students activiely tackled the exercise.

For analysing all textual descriptions of the students' debugging processes in a structured way, we used a detailed list of different criteria. We clustered these criteria into *technical*

TABLE III
FAULTY FORMULA.

Faulty formula	

Code	`for (int i = 0; i < ingredientList.length; i++) {`
	`  weight += ingredientList[i].getWeight();`
	`  calories += ingredientList[i].getCaloriesPerKG() * weight / 10001;`
	`}`

Error Class	STRUCTURAL BLINDNESS
Description	Inadequate formula for computing the overall caloric information of a mixture of several ingredients, which differ with respect to their calories per kilogram and their share in the mixture. Students did not correctly identify the dependencies between the different given data items within the program.
Solution	Carefully read the given task description, identify the different data items and analyze their interrelation; on this basis, model a correct process to compute the calories per kilogram of a mixture of ingredients.

TABLE IV
XOR VERSUS IF/ELSEIF STATEMENT.

XOR	

Code	`if (crtIndexPlus1.charAt(position) == 'O'`
	`  && crtIndx.charAt(position) == 'O') {`
	`  crtCompared += 'O';`
	`} else if (crtIndexPlus1.charAt(position) == 'X'`
	`  && crtIndx.charAt(position) == 'O') {`
	`  crtCompared += 'X';`
	`} else if (crtIndx.charAt(position) == 'O'`
	`  && crtIndexPlus1.charAt(position) == 'X') {`
	`  crtCompared += 'X';`
	`} else {`
	`  crtCompared += 'O';`
	`}`

Error Class	MENTAL TYPO
Description	We see that two conditions are essentially identical (second and third).
Solution	Change the order of the second argument ('O' 'X') in the comparison to 'X' and 'O' or swap crtIndx and crtIndexPlus1 in the third condition.
Error Class	QUALITY GAP
Description	The if/elseif construct represents an XOR operation. Although semantically equivalent, the if/elseif-variant is more complex and error-prone than using the XOR-operator.
Solution	Replace if/elseif construct with the XOR-operator ^ .

criteria as well as indicators for *following instructions* and *writing competence*, thus reflecting a variety of technical and non-technical competencies.

For *technical criteria*, we focused on the following indicators:

- Employing the Wolf Fence Algorithm (rather than uneducated guessing)
- Following a systematic approach
- NOT using anthropomorphic phrases with respect to computer's program execution
- Correct use of technical terminology
- Number of detected defects
- LACK OF weird solutions, such as incorrectly fixed defects (that deteriorated through the "fix")

To compute a score in this area, we simply added up the number of indicators that was fulfilled, leading to a maximum score of 6. Note that the third indicator "number of detected defects" earned a whole point only if *all* the three defects were correctly identified.

To find out whether the students are able and willing to *follow the written instructions* of our teaching unit, we asked them to provide certain information snippets for each of the defects they identified during their bug-detection process. More precisely, for each defect we required the following information snippets:

- Screenshot of variable view in their IDE
- Screenshot of breakpoint as shown in the IDE
- Hypothesis of where the bug could be
  (It was important to have at least one hypothesis. Note that usually, more than one Divide and Conquer step was necessary to detect a bug.)
- Line where student set a breakpoint
- Description of the defect's cause

Furthermore, we requested students to document their name within their solution, and to ensure that screenshots were legible. To compute a score in this area, we added up the number of snippets provided over all three defects. If present, name information and legible screenshots added another two points on this score. This led to a maximum score of 17 points in this area.

To analyze the *writing competencies*, we focus the following aspects:

- Number of typos per amount of written text
- Number of grammar mistakes per amount of written text

VI. Results

When analysing the solutions handed in by our students, we realized that the results of the debugging assignment can tell a lot about our students' skills and competencies, and can even help to uncover some misconceptions. Among others, we investigated correlations of debugging skills with the following base competencies:

- The ability to work in a systematic way
- Reading skills
- Writing skills (i.e. not using adequate technical terms)

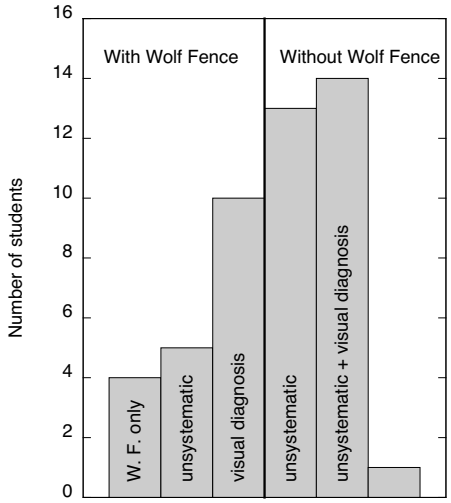

Fig. 2. Different approaches students followed to detect the defects.

- Abstract thinking
- Accuracy

A. Technical competencies

First, let us focus on the results our students achieved in the area of technical competencies. Here, we observed that only four students employed the Wolf Fence approach for defect detection, as requested. Other approaches were visual diagnosis, or some sort of random poking around. Most students used some kind of combination of the following approaches:

1) Application of the Wolf Fence algorithm.
2) Visual diagnosis.
3) Unsystematic poking around to detect the bugs.

Fig. 2 illustrates how many students followed the different approaches.

Defect 2 (Faulty formula) was identified correctly by 47 persons, but incorrectly "fixed" by 14. To a certain extent, this "faulty fix" was somewhat encouraged by our test data, in which the weights of the ingredients summed up to exactly 1 kilogram. However, this "faulty fix" indicates that these students obviously did not properly understand the interrelation of the provided data, thus leading to a faulty formula for calculating the combined calories with respect to the weight of the different ingredients.

Finally, we detected some interesting misconceptions that occured only singularly:

- One person seems to understand breakpoints as some kind of scope. I.e. he always sets one breakopint at the beginning and one at the end of a method.
- One group simply renamed their open office document from `.odt` to `.pdf`, without properly converting its content. This indicates a fundamental problem of coping with IT.
- One team did obviously not discriminate between testing and debugging.

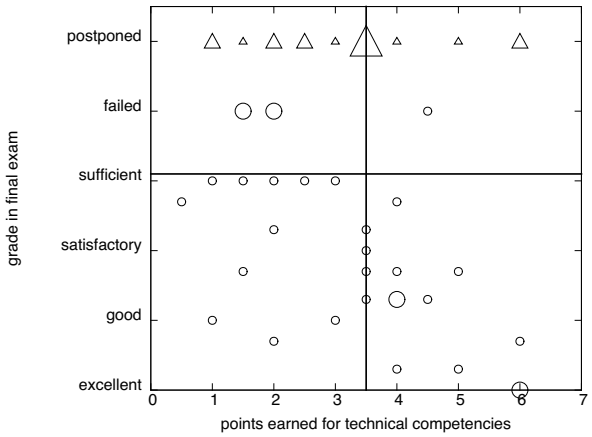

Fig. 3. Scatter plot relating grade in final exam to points earned for technical competencies.

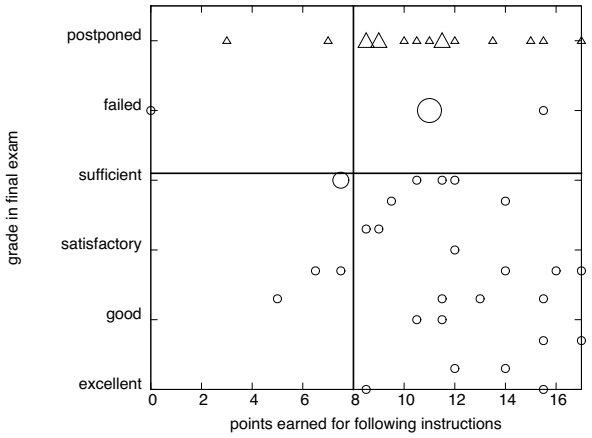

Fig. 4. Scatter plot relating grade in final exam to points received for following instructions.

B. Overalll competencies

As stated above, we believe that the teaching unit can provide insights into non-technical competencies as well as technical competencies. To give a detailled analysis of those base competencies we correlated the score value obtained by every student with the grade they received in the final exam. The latter is a value between *excellent* and *failed*. In addition, we provided the value *postponed* for students that did not even participate in the final exam.

Fig. 3 plots the points earned for technical competencies against the grades students achieved in their final exam. Analogously, Fig. 4 relates points earned for following instructions to the exam grades.

As mentioned before, some students handed their solutions in as a team. Note that some of these teams comprised one rather strong partner and one that had more difficulties when tackling the content alone (see Fig. 5).

In addition, Fig. 6 visualizes the correlation of the ability of using technical terms correctly to the grades that students

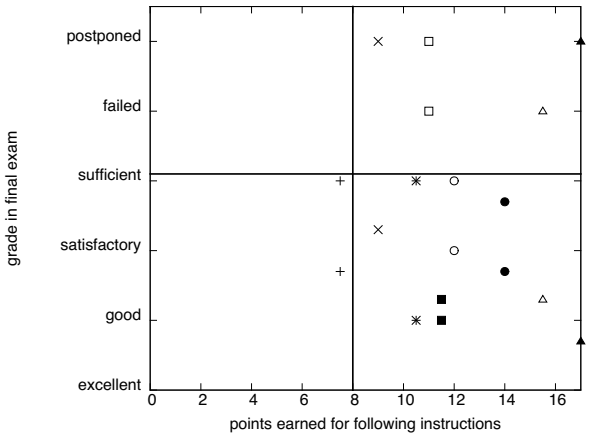

Fig. 5. Relating grade in final exam versus number of points received in the debugging exercise. This plot shows only the results for groups, using different symbols for each group.

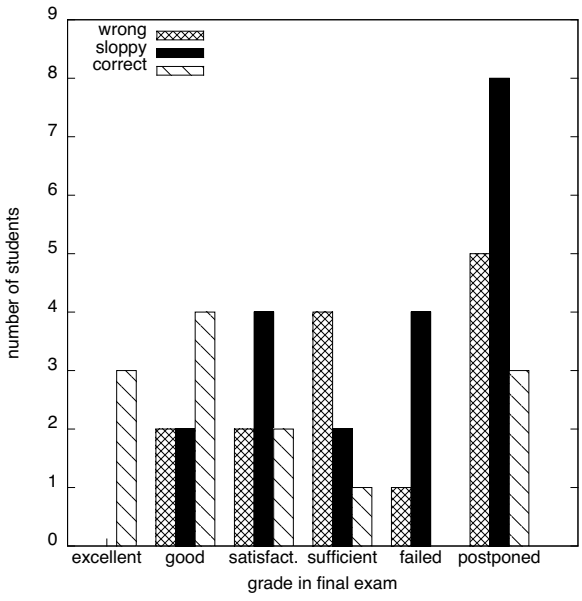

Fig. 6. Visualisation of the correlation between the ability of using technical terms correctly (measured in three steps: wrong, sloppy, and correct) and the grades that students achieved in the final exam.

achieved in the final exam. Here, it becomes obvious that none of the students that used technical terms correctly acually failed the exam!

Finally, we applied Fisher's exact test [19] from statistical test theory to support our observations. Fisher's exact test evaluates whether there exists a significant correlation between two categorical variables. In our case, one variable indicated whether students did (or did not) work systematically. As the other variable, we examined a variety of data, such as:

- Students did (or did not) use technical terms correctly.
- Students did (or did not) make "weird mistakes", such as introducing a "bug fix" that increases the defect rather

TABLE V
2×2 CONTINGENCY TABLE AND RESULTS FOR FISHER'S EXACT TEST FOR SIGNIFICANCE ON INDEPENDENCE OF VARIABLES FROM SYSTEMATIC WORKING.

	working systematically	working unsystematically	
technical terms used correctly	11	4	
technical terms used incorrectly	4	28	$p = 0.0001$
weird mistakes	0	6	
no weird mistakes	15	26	$p = 0.16$
lecturer's solution copied	15	28	
all solutions done by themselves	0	4	$p = 0.2$

than eliminating it.

- Students did (or did not) copy the sample solution provided by the lecturer.

Table V visualizes the results of Fisher's exact test for these three pairs of data. Thus, the hypothesis that "working systematically" is correlated with the students ability to use technical terms correctly, with a very high significance of approximately $p = 0.0001$. For the other two variable pairs, the correlation is somewhat less significant.

VII. CONCLUSIONS AND FUTURE WORK

In this paper we presented our approach for teaching debugging to first-year students. In the teaching unit we designed, students have to find several defects in given code. In contrast to other practical exercises, they were not just asked to turn in code. Rather, they had to hand in a verbal description of the process they applied for finding the defects.

We performed an extensive analysis of our students' solutions, which povided us with an insight into base competencies our students possess in addition to those technical competencies that are usually assessed by typical lab exercises.

Roughly half of our students had difficulties in following the detailed instructions, which included a complete demonstration of the defect detection process for one of the bugs.

An analysis of correlations between several categorial variables considered showed that there exists a strong correlation between the ability to work systematically and a correct usage of technical terms. This shows that basic technical vocabulary is essential in learning software development.

Thus we see on one hand that a course on software development can contain exercises that do not imply the development of code. Those exercises can be a basis for detecting development needs in the area of non-technical competencies.

Furthermore, we were able to detect several misconceptions in some of our students' minds, as well as basic problems of dealing with elementary IT-infrastructure.

In a future version of this exercise we will introduce more bugs, namely one for each of the six categories as stated in [18], addressing different levels of Bloom's taxonomy.

Another idea is to extend the scheme presented in [18] for non technical competencies.

ACKNOWLEDGMENT

This work was supported by the German Federal Ministry of Education and Research (BMBF), grant no. 01PL11025, as part of the "Qualitätspakt Lehre" ("Teaching Quality Initiative") program. Thank you for your support.

REFERENCES

[1] "IEEE standard classification for software anomalies," Tech. Rep., 2010. [Online]. Available: http://dx.doi.org/10.1109/ieeestd.2010.5399061

[2] A. Zeller, *Why Programs Fail: A Guide to Systematic Debugging*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005.

[3] D. Spinellis, "Debuggers and logging frameworks," *IEEE Software*, vol. 23, no. 3, pp. 98–99, 2006.

[4] I. R. Katz and J. R. Anderson, "Debugging: An analysis of bug-location strategies." *Human-Computer Interaction*, vol. 3, no. 4, pp. 351–399, 1987.

[5] D. J. Gilmore, "Models of debugging," *Acta Psychologica*, pp. 151–173, 1992.

[6] E. J. Gauss, "The "wolf fence algorithm" for debugging," *Commun. ACM*, vol. 25, no. 11, pp. 780–, Nov. 1982. [Online]. Available: http://doi.acm.org/10.1145/358690.358695

[7] R. McCauley, S. Fitzgerald, G. Lewandowski, L. Murphy, B. Simon, L. Thomas, and C. Zander, "Debugging: A Review of the Literature from an Educational Perspective," *Computer Science Education*, vol. 18, no. 2, pp. 67–92, Jun. 2008.

[8] D. Barnes and M. Kölling, *Objects First with Java: A Practical Introduction Using BlueJ*, 5th ed. Pearson, 2012.

[9] C.-Z. Yen, P.-H. Wu, and C.-F. Lin, *Engaging Learners Through Emerging Technologies: International Conference on ICT in Teaching and Learning, ICT 2012, Hong Kong, China, July 4-6, 2012. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, ch. Analysis of Experts' and Novices' Thinking Process in Program Debugging, pp. 122–134.

[10] S. Fitzgerald, R. McCauley, B. Hanks, L. Murphy, B. Simon, and C. Zander, "Debugging from the student perspective," *Education, IEEE Transactions on*, vol. 53, no. 3, pp. 390–396, Aug 2010.

[11] "Computer engineering curricula – curriculum guidelines for undergraduate degree programs in computer engineering," 2016. [Online]. Available: https://www.computer.org/cms/Computer.org/professional-education/curricula/ComputerEngineeringCurricula2016.pdf

[12] C. M. Kessler and J. R. Anderson, "A model of novice debugging in lisp," in *Papers Presented at the First Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers*. Norwood, NJ, USA: Ablex Publishing Corp., 1986, pp. 198–212.

[13] V. Thurner, A. C. Böttcher, and A. Kämper, "Identifying base competencies as prerequisites for software engineering education," in *Global Engineering Education Conference (EDUCON), 2014 IEEE*, April 2014, pp. 1069–1076.

[14] P. Nagvajara and B. Taskin, "Design-for-debug: A vital aspect in education," in *Microelectronic Systems Education, 2007. MSE'07. IEEE International Conference on*. IEEE, 2007, pp. 65–66.

[15] G. Novak, A. Gavrin, E. Patterson, and W. Christian, *Just-in-time teaching: blending active learning with web technology*, ser. Prentice Hall series in educational innovation. Prentice Hall, 1999. [Online]. Available: http://books.google.de/books?id=dxdZAAAAYAAJ

[16] L. W. Anderson, D. R. Krathwohl, and B. S. Bloom, *A Taxonomy for Learning, Teaching, and Assessing. A Revision of Bloom's Taxonomy of Educational Objectives*, 1st ed., L. W. Anderson, D. R. Krathwohl, P. W. Airasian, K. A. Cruikshank, R. E. Mayer, P. R. Pintrich, J. Raths, and M. C. Wittrock, Eds. New York: Longman, 2001.

[17] V. Thurner and A. Böttcher, "An "objects first, tests second" approach for software engineering education," in *Frontiers in Education Conference (FIE), 2015. 32614 2015. IEEE*, Oct 2015, pp. 1–5.

[18] D. Zehetmeier, A. Böttcher, A. Brüggemann-Klein, and V. Thurner, "Development of a classification scheme for errors observed in the process of computer programming education," in *International Conference on Higher Education Advances (HEAd)*, 2015.

[19] A. Agresti, "A survey of exact inference for contingency tables," *Statistical Science*, vol. 7, no. 1, pp. 131–153, 1992.