Taking benefit from fellow students code without copying off – making better use of students collective works

Frank Höppner[1]

Abstract: We report about research in progress that aims at generating additional value for programming novices from the collective work of all students in this (and earlier) term(s). We propose live progress reports about the whole course to better estimate the own capabilities. And we propose a solution that autonomously creates hints from other students' solutions to support novices in situations where they do not know how to proceed. A plugin integrates the tool seamlessly in the IDE to avoid additional technical burden. We report first results on a small test group.

Keywords: programming course, code snapshot, live feedback, formative feedback, hint generation

1 Introduction

Introductory programming courses (we consider Java) pose many challenges to novices as they require the development of many skills in parallel. The high number of participants usually prohibits an individual guidance along the way to a solution (e.g. in a programming lab). The advisors are often occupied with approving authorship, while JUnit tests cover functionality (especially in the introductory courses plagiarism checkers may rise too many false alarms for simple exercises). More experienced students are often encouraged to support novices. But true understanding may develop slowly, it needs time. Time is a limited resource for all students, so it seems much more likely that novices are offered a quick repair ("let me quickly fix that for you") or even a complete solution as an illustrative example ("but do not hand in a 1:1 copy"). This kind of help is well-intentioned, but often a small hint or helpful remark ("you are thinking far too complicated, the solution is much simpler") might have been enough to bring them back on track. Besides all that, some novices are simply too shy to ask fellow students (let alone advisors) for help.

This work in progress is concerned about utilizing the collective work of a group of students (the current term and/or previous terms) to provide feedback to individual students autonomously. In every course a number of different students solve the very same exercise. Because of the limited complexity of exercises in introductory courses, almost all reasonable solutions should have shown up after a while. So it should be possible to extract additional value out of this resource. We address the following

[1] Ostfalia University of Applied Sciences, Dept. Computer Science, Am Exer 2, Wolfenbüttel, Germany, f.hoeppner@ostfalia.de

research questions: Is it possible to provide additional value to students in introductory programming courses (Q1) by providing live feedback about the state or progress of the whole group on the current exercise and (Q2) by providing hints about possible next steps by raising questions that were automatically generated from other solutions (but without showing any of the solutions). Of course, we cannot expect hints of the same quality as from experienced advisors, but on the other hand automated hints offer an advantage with respect to availability and anonymity.

2 Related work

Data from coding snapshots has been analyzed before. Often the goal is, however, to estimate the student's performance and to predict final exam grades (e.g. [Bl14]). In contrast, we are not interested in directly estimating the drop-out risk and performance but to provide (automated) feedback (categories taken from the literature review [IV15]).

In [BR06] the authors propose a tool to annotate code with problems or questions, such that a human advisor can provide feedback more efficiently and directly. We aim at something that operates more autonomously. In [An15] reference solutions of the teacher are post-processed by a hint generation tool, which can then reveal parts of the solution successively to the student, depending on their progress. This is related to our intended goal, but again requires manual work and apparently a single reference solution. In our experience, students quite often come up with solutions that deviate somewhat from the intended solution, but are nevertheless valid solutions. It may be annoying to get dragged into a different direction then. Hints were also provided in [DYC10], but the hints directed the students to related tasks and were not obtained by investigating the actual source code, but by finding similarities between the tasks and associations between the covered topics. In contrast, we want to provide hints that are connected to the individual code snapshot and the solution.

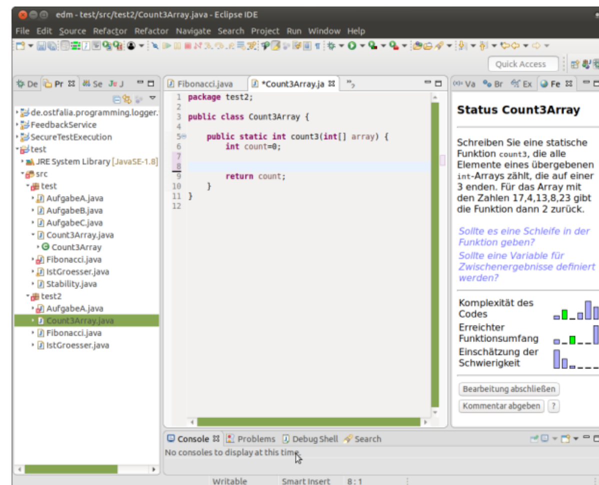
3 Proposed Approach

We have implemented a plugin for the Eclipse IDE that submits the student's current code snapshot to a server (not permanently, only if the plugin has been activated for the current project). Once a student has created a Java class with a particular name, which has been associated with a programming task, the exercise instructions appear within the IDE, which makes exercise distribution and handling more convenient. Using a hashcode on the evolution of the code we can keep track of solutions, but neither need the IP addresses nor a user account, so we have no personalized data about the author (privacy preservation).

Statistics. With every new submission of the snapshot (every time a student saves the file) some statistics are updated and shown to the student. The currently assessed

statistics address code-complexity (weighted sum of instruction types, where the weight becomes larger for more complex instructions types), perceived difficulty (as voted by students when finally closing the exercise), functional completeness (number of passed JUnit tests), duration since first submission (until completion), and the overall code length. From the set of submissions we construct barplots and present them live in the IDE (right view in Fig. 1). The bar that corresponds to the own solution is highlighted. This allows not only to observe the group of fellow students but to roughly know where one fits in this distribution. Some of these statistics are only shown when the exercise has been completed.

Apart from closing an exercise there is only a single further action the students may invoke: Write an *anonymous message* to point out an unclear part of the task description or request a detailed explanation of some concept during the next lecture.



Translation of feedback window (right): Write a static function count3 that counts elements of a passed int-array that end with a 3. Example: 17,4,13,8,24 should yield 2.

Context sensitive hints: Should there be a loop in the function? Should there be a temporary variable for intermediate results?

Bar charts for code complexity, passed JUnit tests, degree of difficulty.

Fig. 1: Screenshot of the IDE. The view on the right shows the tasks, hints, and feedback statistics.

Mining Code Snapshots. The more challenging part is the automatic construction of hints for students who do not know how to proceed or how to attack an exercise. We currently focus on exercises which require the students to write a single function (and test it in the main function). In a nutshell, once a reasonable number of submissions has been received (we used about 20), we (1) extract the core parts of all solutions, (2) group all solutions that follow the same solution path, (3) use the core parts to decide which solution path a partial solution might be following, (4) suggest missing parts as possible next steps. Steps (1) and (2) run offline, (3) and (4) are performed with every new snapshot submission.

To accomplish this, each submission is processed by a parser and the resulting abstract syntax tree is turned into a graph (Fig. 2) by merging nodes that refer to the same variable (e.g. from declarations, expressions or assignments). Every variable usage

therefore establishes a path from the respective instruction to the used variable. Other transformations are applied to the graph, for instance we flatten expression trees to become less sensitive to the exact typing of expressions (e.g. changed evaluation order by using parentheses).

```
public class AufgabeB {
  public int count3(int[] a) {
    int ctr = 0;
    for (int i=0;i<a.length;++i) {
      if (a[i] % 10 == 3) ++ctr;
    }
    return ctr;
```

Fig. 2: Graph constructed from top right code snapshot; green nodes match a reference solution

We then identify all *paths* in the graph leading from a variable to the *function declaration* and use them as the core parts of a code snapshot. These paths collectively capture how all variables are used and therefore represent a footprint of the solution. We thus intentionally exclude literals or operators by the aforementioned transformations. These details are relevant to obtain correct results and to pass the tests, but many different expressions may lead to the same results. Including exact expressions in the solution paths would require a much larger collection of code snapshots. So we focus on helping the students to arrive at a valid instruction skeleton.

When comparing a student's solution against the identified reference solutions (step (3)), we identify these core paths and determine how many paths of a valid solution have already been found. We take these counts as evidence which solution the author may be aiming at. The green nodes in the graph of Fig. 2 show which part of the graph has been matched to a selected instance of reference solutions. Finally, missing paths indicate that certain structural elements are missing in the solution. We transform a missing path into questions in natural language (step (4)), for instance "Should there be a loop in the function?" or "Should there be a temporary variable to store intermediate results"? We phrase the hints as questions rather than giving instructions for two reasons: Firstly, we want the student to think about it rather than blindly following instructions. Secondly, at some occasions the hints may not be correct, because the student follows a different plan than the program is assuming. In the latter case we prefer that the student sticks to her plan rather than feeling advised to change it.

4 Evaluation

As this is still work in progress, we can provide a preliminary evaluation only. A group of n = 32 students has tested the plugin and provided feedback by means of a questionnaire that included free text questions as well as expressions of dis/agreement on a Likert-type scale (1=totally agree, 5=totally disagree). The results are shown in Fig. 3.

The students particularly appreciated the tight integration into the IDE (Fig. 3(a)). In the past they all have experienced a somewhat annoying process of submitting solutions to some test facility where the JUnit tests get executed (e.g. copying and pasting it into a learning environment, or submitting it to a version control system). Struggling with the programming language and the IDE already, novices typically do not ask for additional technical burden of such kind. The possibility of asking questions anonymously (Fig. 3(b)) was also highly appreciated (under the assumption that the instructor actually addresses these questions in the lecture, as some students honestly and correctly remarked).

Questions (c) and (d) asked for the utility of the live feedback (live comparison of my own code with that of my fellow students) and the long-term statistics (comparison of my own code against a long-term collection). Overall, the students appreciated both ways of feedback (much more rated ≤3 than ≥3), but from the written remarks the students were really divided. Some were eager to compare themselves with fellow students in the same course and did not care about former terms (they appreciated a kind of live challenge), others were more reserved because this may give novices a negative feeling of underachievement and instead saw an advantage in comparing themselves to a more representative and larger sample. In a future version, we may let the students decide on their own which type of statistics they prefer to see or provide a stacked barplot with different skill levels. Most of the presented individual statistics were very much appreciated (Fig. 3(f-k)), only the editing time was found to be misleading as it included finished as well as just started exercises at the same time. Here, only the duration of finished exercises should be reported.

The guiding questions for novices were also very much appreciated (Fig. 3(e)). We received remarks such as “The hints make me think about the problem, but do not tell me directly what to do.”, which is very much what we wanted to achieve: We did not want to imitate the IDE’s auto-completion functionality, which can be applied thoughtlessly and therefore does not encourage students to think about the exercise and their code. Others replied that they found the hints encouraging and that they reduced the feeling of being left alone with the tasks. But it needs a more detailed evaluation in the future to figure out to what extent the hints can actually enable students to solve tasks.

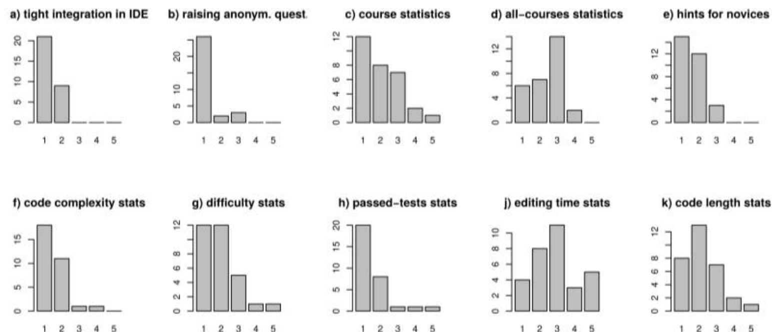


Fig. 3: Answer on a Likert-type scale: Did you find ... helpful (1) or unhelpful (5)?

5 Conclusions

Students work alone or in pairs on programming exercises in introductory courses, but usually cannot objectively compare themselves to the course performance. Getting some *live feedback* about the whole course (e.g. “How many tests did others pass by now?”) was considered useful and motivating (which answers (**Q1**) positively), although the respondents were divided whether they want to compare against their own course only or all former courses. Regarding (**Q2**) the fully automatically generated hints, that came in the disguise of questions, were also appreciated and considered helpful for novices who struggle with the next steps of their solution – even though the hints did intentionally not convey too much of the solution. We consider this preliminary evaluation as encouraging and will continue to improve the system and its evaluation.

Bibliography

[An15] Antonucci, P. et al.: An incremental hint system for automated programming assignments. In: Conf. Innovation and Technology in Comp. Sc. Education, ITiCSE. pp.320–325, 2015.

[Bl14] Blikstein, P. et al.: Programming Pluralism: Using Learning Analytics to Detect Patterns in the Learning of Computer Programming. Journal of the Learning Sciences, 23(4):561–599,2014.

[BR06] Bancroft, P.; Roe, P.: Program annotations: Feedback for students learning to program. In: Australasian Computing Education Conf. pp. 19–23, 2006.

[DYC10] Dominguez, A. K.; Yacef, K.; Curran, J. R.: Data Mining for Individualised Hints in eLearning. In: EDM. pp. 91–100, 2010.

[IV15] Ihantola, P.; Vihavainen, A.: Educational data mining and learning analytics in programming: Literature review and case studies. Proc ITiCSE Working Group Report, pp. 41–63, 2015.