Monitoring Students' Mobile App Coding Behavior

Data Analysis Based on IDE and Browser Interaction Logs

Markus Fuchs[1], Markus Heckner[2], Felix Raab[2], Christian Wolff[2]
[1]Information Science Group / [2]Media Informatics Group
University of Regensburg
Regensburg, Germany
{markus-bernhard.fuchs, markus.heckner, felix.raab, christian.wolff}@ur.de

***Abstract*—This paper describes a case study of assessing student's coding behavior and skills in a realistic development setting. Students had to solve typical programming problems in the context of app development for the Android platform using the Eclipse IDE. Data was analyzed using IDE as well as browser interaction logs. In addition, screen recordings of the students' interaction with the IDE provide further insight. In this paper we present the first results of our ongoing work.**

Keywords—android; coding behaviour; alternative learning assessment; IDE interaction

I. Introduction and Motivation

We present data gathered in an application programming course for undergraduate media informatics students. The media informatics B.A. degree program was introduced at Regensburg University in 2010. In this B.A. program, students typically study two major subjects. While many different combinations from more than 40 subjects are available, almost two thirds of students study media informatics in combination with information science with both subjects having a curricular agenda close to applied computer science.

Software development as well as software engineering play a major role in this degree program, especially focusing on interactive systems and digital media. Data for the work presented here was collected during an application programming course in the winter term 2012 / 2013. Participating students typically are in their second or third semester in the media informatics degree program.

Students already have substantial knowledge in object-oriented programming using the Java programming language, as the application programming course is a follow-up class to an introductory object-oriented programming course taught using Java. In the application programming course, the major goal is to reach a good understanding of typical application programming issues like network and database connectivity, user interface programming or data and file management. We use Android and the field of interactive mobile applications on smartphones [32]. The students learn basic concepts of the Android operating system as well as using typical software development tools like current IDEs. After a phase of teaching Java as well as application programming with reduced learners' IDEs like BlueJ (http://www.bluej.org/, [4]) we have been using Eclipse (http://www.eclipse.org) as the standard IDE for several semesters.

As in many course settings, our assessment of learning outcomes was based on programming assignments students had to complete during the semester and a final written exam. However, this has the following downsides: First, a written exam can never emulate a real programming task, especially when using a complex framework such as Android, and is therefore always an artificial situation. Second, instructors can not assess whether students have completed their work on their own or if they consulted fellow students from advanced semesters or other sources. Thus, they miss essential problem solving experience and coding practice and potentially are not able to reach the learning goals. We were looking for a scenario that allows for directly observing the students' programming behavior and problem solving strategies.

The paper is organized as follows: Ch. II gives a short overview of related work. In ch. III we explain the basic technical setup for our study, followed by a description of the programming tasks used during the exam (ch. IV). Chapters V – VII discuss the data we have collected, focusing on IDE usage data (ch. V), an analysis of screen recording videos taken during the exams (ch. VI) as well as web browser log analysis (ch. VII). A short conclusion finalizes the paper.

II. Related Work

Recently, teaching programming has picked up the trend for mobile computing [12], [32]. As the Android mobile platform is based on the Java programming language, it may be used for introductory as well as advanced programming classes [27]. Although it is clear that engineering mobile apps poses specific software engineering challenges [34], availability of mobile devices as well as their sensor rich technical environment make them an attractive development platform [8] which may not only be employed for GUI development scenarios, but also for understanding event-based programming and automated testing [3].

[25] give an overview of early programming behavior studies. Programming behavior has been analyzed using various types of data, including video analysis [19], [35], analysis of graded programming assignments [11] or students' newsgroup discussions [18]. [29] present a classification of source code search strategies.

[2] show that typical behavioral patterns exist and dedicated models of how students learn to program have been proposed [9], [26]. Several studies have been dedicated to low level IDE

interaction analysis: [37] offer a classification for IDE interaction. [20] and [21] describe copy & paste strategies employed by IDE users. General aspects of IDE usability have been described by [22], while [30] suggest usability improvements for the Eclipse IDE. [10] establishes programming success predictors on the basis of IDE usage.

Programmers' web search behavior has been studied by [17] using a specific software interfaces [31]. Usage of dedicated Q&A sites for software developers has been studied by [24] and [33]. [16] present *HelpMeOut,* a social recommender system that helps with compiler error messages.

Bringing together IDEs and relevant content from the web as an aid for programmers has been suggested repeatedly ([5], [13], [14]). Being able to pick up material from the web in an opportunistic mashup style has been described by [15]; similarly, [6] describe how programmers "opportunistically interleave Web foraging, learning, and writing code" ([6], p. 1).

III. Technical Setup and Tools

The assessment was conducted in a university facility that is equipped with 57 personal computers running Windows and Linux operating systems. At each assessment, all students were divided into two consecutive groups in order to alleviate both technical and organizational supervision concerns. Students were provided with a preconfigured version of the *Android Development Tools* (*ADT*, http://developer.android.com/tools/sdk/eclipse-adt.html). *ADT* is an *Eclipse* plugin but also exists as standalone version preconfigured with all necessary tools for *Android* development. Since all students had used *ADT* for course assignments before, they were familiar with the features of the development environment. Installation of such tools in university computer labs is generally easier using self-contained packages with no external dependencies so that teaching staff is able to update and configure all tools without involving technical support. In addition to *ADT,* we have preinstalled tools for logging and analysis of coding behavior at three different levels which we will describe in the following.

A. IDE Events

We used the publicly available *Eclipse* plugin *Fluorite* [37] that logs interaction events when working with the development environment. Compared to other logging tools, this plugin captures low-level commands in the editor so that researchers can explore frequently triggered commands or detect fine-grained usage patterns. The data is logged into *XML* files each time the *IDE* is closed. In order to prevent data loss, we asked students to restart the development tools before starting work on a new task. The authors of the tool also built a separate application for producing analysis reports and charts. However, we have found it more useful to extract relevant information from parsing the *XML* files with custom scripts we have developed for our analysis tasks. For future setups, we would consider adding another event logging plugin since *Fluorite* does not capture some details that appear interesting for closer inspection, for instance: Users select particular *menu items,* interact with the *line gutter* or produce *syntax errors* and cannot test the application. Gathering data about such interactions could reveal potential usability issues and help refining existing tools.

B. Browser Logs

Since students were permitted to use code examples from their course assignments and to search the web for answers to problems, we also logged certain browser events. We included a portable version of the *Firefox* browser into the software package and asked students to use this browser instead of the standard browser of the operating system. An internally developed *Firefox* plugin logged data into a *JSON* file for each user, including events such as opening, closing, and switching of browser tabs, or selecting and copying text. This data was used to identify search terms and frequently visited web sites when students look up documentation or try to solve a coding problem. Furthermore, by logging text selections and copy commands, it can be determined which code snippets were actually pasted into the editor.

C. Screen Recording

Although logging *IDE* and browser events is useful for quantitatively extracting information about certain aspects of coding behavior, we have found it essential to include screen recording into our software package. For example, students' intentions when they are trying to solve a problem over a period of several minutes are difficult to detect by relying solely on interpreting the low level events given in interaction logs. Watching the corresponding video, however, in many cases makes the nature of programming problems quickly apparent. What high-level steps were carried out by the students in order to find the solution may be seen as well. Although the process is time-consuming and we have not yet been able to fully analyze all of our collected video material, we have already found interesting behavior by randomly picking some videos and watching only few minutes. All videos were captured using a modified command-line version of the freely available screen recording software *CamStudio* (http://camstudio.org/).

Automated Setup and Submission

All of the tools mentioned above were integrated into one single package that was distributed over the local network before the assessment started. Students were then asked to execute a *setup script* that created shortcut icons for the IDE and browser and started the video recording. For submission of the source code, log files, and video file, another script compressed all data into a single file. This file was then manually collected from each student. The reasons for manual collection were partly due to file size restrictions of our technical infrastructure since video captures produced several gigabytes of data for each participant.

Prior to the analysis students had to sign a consent form that data is collected in an anonymized form and used for research purposes. If students voted not to sign, analyses would have been turned off. However, all students agreed to take part in the study.

IV. Programming Tasks

Since the time of the exam was limited to 120 minutes, students did not have to code complete apps from scratch, but to extend or refactor existing code. All code was provided as an

Eclipse package that the students could import into their workspace.

The following list of tasks the students had to complete constitutes a whole exam:

1) *QuizApp* - Create a small quiz app that presents a question to players. The players can guess "quietly" for themselves, i.e. without making any input and check if their guess is correct once they press a "solution" button. The questions and the answers are integrated as XML string resources in the starter project. Furthermore, the complete game logic classes (i.e. the model in object-oriented terms) were already provided. Students had to correctly instantiate those classes, create the user interface (UI) and wire the model and the UI together.

2) *Responsive User Interfaces* – all Android applications possess a single application thread by default. Parallel execution needs to be implemented by the programmer using Threads or "AsyncTask", an approach specific to the Android framework. The starter project contains an application that calculates the faculty of a fixed number. This calculation takes several seconds to complete and consequently freezes the UI. Students had to add asynchronous functionality to the app and move the time-consuming calculation there, so that the UI stays responsive during calculation and is correctly updated using an alert dialog after the calculation has finished.

3) *Refactoring* – from the first course in object oriented programming onwards, students are introduced to well established coding practices. Among many others, these practices include variable and class naming conventions, class vs. instance variable usage, the model-view-controller pattern (MVC), as well as basic design principles like information hiding. The goal of this task was to refactor the code of an application which violates some of these principles, without altering the functionality.

V. IDE USAGE AND INTERACTION

Adopting IDEs for teaching software engineering poses a dilemma for teachers: Introducing complex software such as *Eclipse* enables students to use the same tools as professionals but results in steep learning curves and requires additional supervision. For instance, we have observed issues with students not being able to install *Eclipse* on their own machines due to various compatibility issues or other obscure errors. Introducing special IDEs for novices may generally ease usage but also diverges too much from development as practiced in industry. As far as development for the *Android* platform is concerned, one example of an alternative programming environment is *Android App Inventor* [1]. Although working *Android* applications are created as output, the development process is drastically different since the tool uses a graphical approach: Programs are created by wiring and configuring multiple program blocks instead of writing code from scratch using the *Android* framework.

While both approaches have strength and weaknesses, from a teaching perspective, knowledge about actual usage of the programming environment is useful as students can be made aware of important functionality that they might have missed. Moreover, differences between experienced and inexperienced participants become apparent when they attempt to solve problems by using (or *not* using) IDE features that are supposed to support developers. In our study, we have found examples of more experienced students applying *help tips* or *quick fixes* in the *Eclipse* IDE more often and more successfully than participants that produced wrong solutions. It has been shown that the use of certain IDE features significantly correlates with final exam grades [10]. At the same time, the fact that particular functionality is *not* used might indicate lacking competencies beyond IDE usage.

From the perspective of toolmakers, recognizing patterns and deeper understanding of commands used is a precondition for designing improved tools. Since IDEs are often too functionality-oriented and fail to address important usability problems, they do not always hold the promise of improving developer productivity [22]. Knowledge about users' interaction can inspire new software development tools aimed at better supporting both experts and novice users, who are particularly hindered by usability issues in terms of control and efficiency [22].

IDE Log Analysis

In order to examine students' interactions in the IDE, we extracted and merged all XML log files of the *Fluorite* [37] plugin so that particular events could be analyzed in its entirety or per participant. On the one hand, investigators can get insights about high-level information such as different types and frequencies of executed commands. On the other hand, detailed analysis of the interaction of multiple low-level events also reveals low-level usage patterns [37]. In this work, we report some preliminary results of extracting command frequencies from the log files by our custom analysis scripts.

Tab 1. shows average command frequencies per participants for a number of selected commands. The table does not include the most frequent commands, such as inserting text, selecting text, or moving the caret as they do not convey relevant information on programming behavior. One noticeable result is the average number of unique commands trigged per participant. Compared to the vast number of available commands in *Eclipse,* this value suggests that users are familiar only with a very small and limited subset of actions. Refactoring commands remain mostly unused, although one of the tasks in the first assessment encouraged students to improve the quality of existing code and fix bad coding style. In some of the video captures, we could also identify low productivity and error-proneness because those operations were often performed manually instead of relying on IDE support.

The high number of opened files indicates numerous context switches, despite projects consisting of only two or three main files that had to be edited in order to complete the task. As already known from other studies, the *backspace* key is one of the most frequently executed keystrokes when editing code, typically used to fix typing errors or rename elements [37].

This casts doubt on the usefulness of written tests for computer science courses where students still have to solve programming problems on paper. In fact, students' solutions have often been indicative of frequent revisions or illegible solutions, which complicates grading for teachers. *Auto-complete menus* (Content Assist) were often triggered manually, whereas video analysis has revealed that proposals displayed by the IDE were not always executed and participants instead resorted to consulting documentation in the web browser.

TABLE I. ECLIPSE COMMAND FREQUENCIES

Type	*Average*
All Commands Used	**334.6**
Unique Eclipse Commands Used	**14.0**
Delete (Backspace)	135.2
Files Opened	91.1
Manual Content Assist	*65.7*
Save	*31.8*
Paste	*26.1*
Copy	*15.3*
Cut	6.2
Manual Quick Assist	5.5
Undo	5.1
Organize Imports	4.2
Code Formatting	3.4
Rename Refactoring	2.3
Redo	0.3

VI. VIDEO-BASED CASE STUDIES

In the following sections, we briefly present two case studies of students trying to solve one of the tasks of the first assessment (see section IV for a description of the task). Their applied strategies in solving the problem differ significantly with respect to IDE usage and adaptation of example code from different sources. The first student could be regarded as more experienced, while the latter is more inexperienced and received worse grades in two introductory computer science courses. Although we do not claim that the students' behavior is generalizable, we have found interesting features that might be repeatedly shown by other students within comparable skill levels.

A. Case 1

The student starts by first inspecting the *Android layout* file (an XML file containing user interface components). He then creates references to user interface components in the *activity* file (the main code file for the application) and adds events that respond to user interactions (e. g., clicking on a button). Since from the task description he knows about the requirement of *threading*, he searches for "android threading" in Google, follows the first result that links to the Android documentation about processes and threads, and studies the example code. Based on the code examples, he creates the required *AsyncTask* class and uses *help tips* and *quick fixes* in Eclipse to complete the class implementation without copying from the documentation. Next, he attempts to use *auto-complete proposals* for adding a suitable method call to show the required progress bar component but cannot find a matching method. He then performs a series of Google searches ("android progressbar move on asynctask"), opens the first search results in multiple browser tabs, and studies code on the QA website *StackOverflow*. Additionally, he consults API guides in the *Android* documentation and uses local search. After another Google search for "spinning progressbar android", he opens the first few search results (*StackOverflow*) in new browser tabs. He then switches back and forth multiple times between code examples and the editor window, trying to add the correct implementation. A similar process could be observed when he had worked on showing the alert dialog component (search for "alertdialog android", opening browser tabs, switching back and forth between code example and editor). After over half an hour of editing code, he tests the application for the first time and notices that the alert dialog is not shown. He continues by matching his code to example code from documentation and various opened websites, reads *help tips* and *warnings* in *Eclipse*, and tests the application after each change. Finally he discovers the missing method call to actually show the alert dialog and successfully completes the task. In total, he produced about 35 lines of working source code in about 45 minutes.

B. Case 2

The student also starts by inspecting relevant project files and adding events in order to respond to user interface interactions. He performs the first Google search on how to show the alert dialog component, opens the first results in new browser tabs, and implements the dialog based on example code. After about 20 minutes, he tests the application and checks the working dialog component. He continuous by trying to implement *threading* based on code of previous assignments, which he opens in a separate text editor window, and pastes copied methods into the editor. From then on, the student tries several different approaches to implement asynchronous background processing but fails each time, although he consults various websites and his own code examples. This is mainly because he does not recognize the required structure and execution order of methods. He fails to understand which methods initialize, perform, and finalize the background task. The examples partially contained additional code that the original author had added for demonstration purposes. This code was then copied without adapting it, causing errors in the student's program. After several additional web searches, inspecting example code, and changing parts of the code, the student finally gives up and continuous with the next task. In total, he produced about 95 lines of erroneous source code in about 75 minutes.

C. Discussion

Besides the first student successfully completing the task in less time and with less code, there are several observable differences between both students. In case 1, web search is more focused and directed towards specific problems. He also seems more familiar with the *Android* documentation and API guides. Example code is not copied but first studied and then

adapted to the requirements of the task. Furthermore, the student deliberately and often uses supportive *Eclipse* features such as *help tips*, *warnings*, *quick fixes* and *auto-complete*. The second student, on the contrary, less frequently and less actively uses supportive *Eclipse* features. It could be argued that this somewhat confirms study results where IDE usage is correlated with exam grades [10]. The second student's approach could be described as trial-and-error-programming. He also relies on web search and examples but fails to reshape the code to solve the task. Moreover, he is misguided by example code since he fails to understand core concepts and consequently does not identify code that was added for demonstration purposes only. Another remarkable difference is variable naming: In case 2, there is no clearly recognizable naming scheme, different conventions and styles are randomly mixed and inconsistently applied throughout the program.

In general, the observations mentioned above confirm the usefulness of such assessments since students obviously cannot solve programming tasks by simply copying from example code but rather need to fully understand the underlying core concepts.

VII. Browser Log Analysis

To track the usage of the portable installation of the Firefox internet browser during the exam, we developed a dedicated add-on with the Jetpack SDK[1]. This SDK provides multiple high-level APIs that enabled us to easily track and log multiple browser-related events to a local JSON file:

- *open:* Triggered when a new browser tab was opened.
- *ready:* This event was triggered every time the document object model (DOM) of a website was loaded. The add-on then saved the URL of the loaded page. Page loads that Firefox could serve from the cache were not logged, though. This means that there were no events triggered when the user hit the browser back button, for example.
- *activate:* Every time the user activated a previously inactive tab, the add-on saved the URL of the page loaded in this tab.
- *selection:* When the user selected text on a page, the add-on logged both the selected text and the URL of that page.[2]
- *clipped:* By injecting a script in every loaded website, the add-on listened for the keyboard shortcut CTRL+C and logged the content of the clipboard and the source URL of the copied text.
- *closed:* Triggered when a browser tab was closed.

In total, our add-on logged 1942 events (Tab. 2 shows event frequency by type) in the exam of the winter term 2012. The event counts per student lie between 14 and 266 and can serve as an indicator for usage intensity. So, every student took advantage of the opportunity to use the internet browser during the exam at least once, but the intensities differ starkly.

TABLE II. Event Frequencies

Event Type	*Count*
ready	979
activate	418
closed	286
open	223
clipped	36

To find out what the students mainly used the browser for, we took a closer look at the logged URLs for all 'ready'-events. When opening a search result, Google redirects the browser over a special referrer URL. Because of that, there were always two events in the dataset when the student clicked on a search result link: One for the referrer URL and another one for the real URL of the search result. After removing these duplicated events, 58 % of all ready events were generated either by directly opening the Google homepage, by opening a Google query through the browser search bar, or by clicking a link in the search results page. Another 25.2 % of the ready events arose through various "about"-pages the Firefox browser loads when opening a new tab, for example. Only the remaining 16.8 % of the URLs were loaded either because the student directly typed it in the address bar or clicked on a link within a page that was opened via the search results list. So, the students used the internet browser mainly for Google searches, opened on average about two results per search and only rarely navigated to another page from there.

Next, we analyzed the 140 Google search queries our add-on captured. Seven of them were neither development- nor task-related and were removed from the dataset. The remaining queries were manually put in at least one of the following seven categories, based on the taxonomy suggested by [13]. When the search was ambiguous we turned to the videos for clarification.

- *Code elements:* Queries that did contain at least one API element even if the spelling was not correct, e.g. "android alertdialog", "asynch task", "asynctask progressbar".
- *Code ideas:* Queries where the student knew what he wants to do but had not already identified the corresponding API element. Examples: "invisible android text", "android string array aus xml" (Engl. "android string array from xml")
- *Sites:* When the student searched for a specific site or used the site name as filter for a code elements search. Examples include "android developer", "dicct", "android api", "stackoverflow android alertdialog".
- *Troubleshooting:* Queries where the student apparently looked for solutions to a specific problem or that did contain an error message. Example searches include "android app not responsive", "cant set asynch

[1] https://wiki.mozilla.org/Jetpack.

[2] Probably because of SDK stability issues, there were no selection events logged in the dataset from the winter term 2012.

task public", "The constructor AlertDialog.Builder (new View.OnClickListener(){}) is undefined".

- *Basic Programming:* Searches where the student looked for information about a basic programming concept not specifically related to the Android API, e.g. "auf methoden aus anderem package zugreifen" (Engl. "access methods of another package") or "android statische methoden aufrufen" (Engl. "android call static methods").
- *Concepts:* Examples for queries in this category are "refactoring", "sauberen code schreiben" (Engl. "write clean code"), where the informational goal was at a higher level than code elements.
- *Tools:* Queries where the student looked for information about the available tools, e.g. "eclipse show line number".

As Tab. 3 shows, students mostly used search to overcome what [23] call *use* or *coordination* learning barriers: They knew *which* API elements they had to use to solve a problem, but not *how* to use them. This comes as no surprise, as students get deep insight into Android application development during the course. Therefore, they should be familiar with the API elements they had to use for the tasks of the assessment. The relatively high number of searches for specific sites can be explained with the fact that the students could not use their own browser, with the consequence that they had no access to their bookmarks. The sites the students opened this way were the Android developer homepage[3], an online dictionary[4], the course's eLearning platform[5] and the university homepage[6].

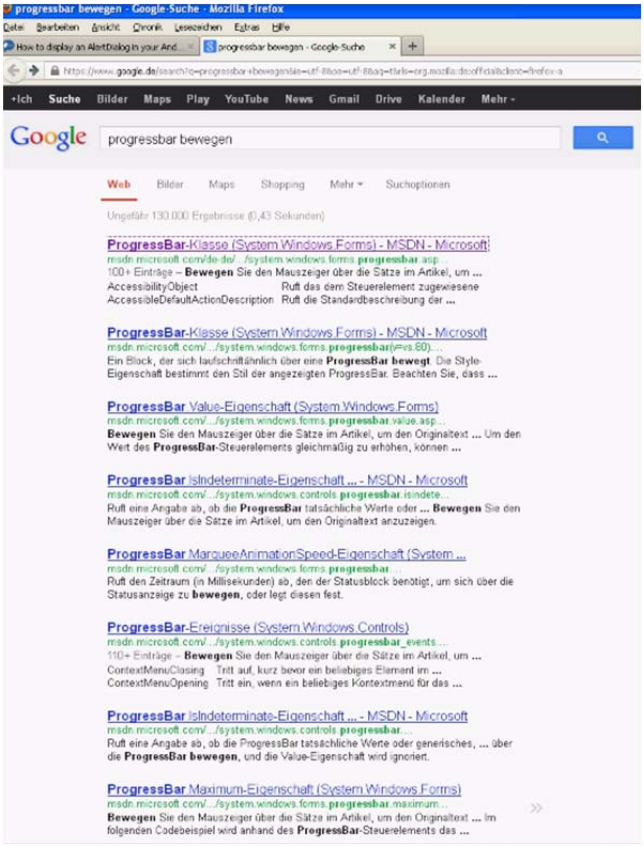
TABLE III. CATEGORIES OF QUERIES

Category	*Percent*
Code elements	60.9
Code ideas	14.3
Sites	13.5
Troubleshooting	6.8
Basic programming	3
Concepts	2.3
Tools	1.5

It is also interesting to note that more than two thirds of all queries contained the term "android". Apparently, students used it as filter to remove results for code elements with similar names from other platforms (e.g. Windows). The importance of extending the search query with the term "android" becomes particularly clear in one case (depicted in Fig. 1), where the student searched for "progressbar bewegen" (Engl. "progressbar turn") and the first eight results all pointed to the MSDN documentation[7].

[3] http://developer.android.com
[4] http://www.dict.cc
[5] http://grips.uni-regensburg.de
[6] http://www.uni-regensburg.de
[7] http://msdn.microsoft.com

Fig. 1. Search results for a query where the term "android" was omitted.



The Google search result referrer URL not only contains the URL of the target page but also the rank it had on the result list. Unsurprisingly, more than 90 percent of all opened results were positioned at one of the first five ranks.

TABLE IV. TOP 5 RESULT DOMAINS

Category	*Percent*	*Count*
stackoverflow.com	40.7	105
developer.android.com	19.8	51
www.mykong.com	6.2	16
www.helloandroid.com	4.7	12
msentwicklung.blogspot.com	2.3	6

Finally, we examined the target pages of all opened search results (Tab. 4 shows the Top 5 domains of opened results). One immediately notices the predominance of the Q&A site StackOverflow[8]. It clearly shows that "Google is UI" was one of the building blocks of this site and, from the beginning, it was designed with search engine optimization in mind [33]. The questions at StackOverflow generally are focused on a very specific problem or task, e.g. "How to call a Java class from a Android Activity"[9], "Android: AsyncTask recommen-

[8] http://stackoverflow.com
[9] http://stackoverflow.com/questions/6093594/how-to-call-a-java-class-from-a-android-activity

dations: private class or public class?"[10], "Android: 'thread exiting with uncaught exception (group=0x4001d88)'"[11]. This helps us explain why the students accessed 87 different questions in the 105 results, of which 75 were accessed only once. As the question sites typically contain lots of source code examples, a good overview of how to use a specific API element can be obtained after viewing some of the related questions. This may be another explanation for the high popularity of StackOverflow results during the exam.

VIII. Conclusion and Outlook

The results given above represent first insights from selected data only. Especially the extensive video material calls for an in–depth analysis of programming behavior and students' problem solving strategies which we want to perform in the near future. The fact that a more or less arbitrary selection of sample material as presented above already yields some relevant insights makes this a promising perspective, all the more as only few video-based studies on programming behavior exist so far [19], [35]. The combination of IDE and browser logs with video analysis provides both, low level interaction details as well as high level insight into typical strategies and problems the students encounter.

We believe that the proposed approach offers a realistic environment to assess student's programming performance and thus to determine whether learning goals were reached. Our first qualitative judgments reflect tremendously improved competence levels of students when compared to earlier courses with only homework assignments and a traditional exam. However, this needs to be verified in additional studies. We can only speculate if the daunting task of live coding creates a very strong incentive for thorough coding practice and preparation.

Additionally, the effort required for setting up the coding environment and gathering results by far exceeds the effort for a traditional assessment such as an exam, because special computer labs need to be organized, the software needs to be deployed and results need to be gathered directly after the exam. We plan to streamline the rollout process and the results gathering.

In a more general perspective, the study design proposed here might also be used for longitudinal studies of students' programming behavior. Given the students approval, long time behavioral changes reflecting respective changes in programming competencies could be analyzed using log as well as video data as described above. Such data might be used for individual counselling as typical problems like not using helpful IDE features are detected from the log data automatically. In the long run, such data might also be used for enhancing IDE help functions even further.

[10] http://stackoverflow.com/questions/4823891/android-asynctask-recommendations-private-class-or-public-class

[11] http://stackoverflow.com/questions/7740767/android-thread-exiting-with-uncaught-exception-group-0x4001d800

References

[1] Ahmad, K., & Gestwicki, P. (2013). *Studio-based learning and app inventor for android in an introductory CS course for non-majors*. Paper presented at the Proceeding of the 44th ACM technical symposium on Computer science education, Denver, Colorado, USA.

[2] Allevato, A., & Edwards, S. H. (2010). *Discovering Patterns in Student Activity on Programming Assignments*. Paper presented at the The Engineering Educator of 2016. 2010 ASEE Southeast Section Conference, Blacksburg, VA. http://se.asee.org/proceedings/ASEE2010/Papers/PR2010All158.PDF

[3] Allevato, A., & Edwards, S. H. (2012). *RoboLIFT: engaging CS2 students with testable, automatically evaluated android applications*. Paper presented at the Proceedings of the 43rd ACM technical symposium on Computer Science Education, Raleigh, North Carolina, USA.

[4] Barnes, D. J., & Kölling, M. (2011). Objects First with Java: A Practical Introduction Using BlueJ: Prentice Hall Press.

[5] Brandt, J., Dontcheva, M., Weskamp, M., & Klemmer, S. R. (2010). *Example-centric programming: integrating web search into the development environment*. Paper presented at the Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, Atlanta, Georgia, USA.

[6] Brandt, J., Guo, P. J., Lewenstein, J., Dontcheva, M., & Klemmer, S. R. (2009). *Two studies of opportunistic programming: interleaving web foraging, learning, and writing code*. Paper presented at the Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, Boston, MA, USA.

[7] Buckley, J., Exton, C., & Good, J. (2004). *Characterizing Programmers' Information-Seeking during Software Evolution*. Paper presented at the Proceedings of the 12 International Workshop on Software Technology and Engineering Practice.

[8] Dabney, M. H., Dean, B. C., & Rogers, T. (2013). *No sensor left behind: enriching computing education with mobile devices*. Paper presented at the Proceeding of the 44th ACM technical symposium on Computer science education, Denver, Colorado, USA.

[9] Dagdilelis, V., Satratzemi, M., & Evangelidis, G. (2002). What they really do?: attempting (once again) to model novice programmers' behavior. *SIGCSE Bull., 34*(3), 244-244. doi: 10.1145/637610.544513

[10] Dyke, G. (2011). *Which aspects of novice programmers' usage of an IDE predict learning outcomes*. Paper presented at the Proceedings of the 42nd ACM technical symposium on Computer science education, Dallas, TX, USA.

[11] Edwards, S. H., Snyder, J., Pérez-Quiñones, M. A., Allevato, A., Kim, D., & Tretola, B. (2009). *Comparing effective and ineffective behaviors of student programmers*. Paper presented at the Proceedings of the fifth international workshop on Computing education research workshop, Berkeley, CA, USA.

[12] Goadrich, M. H., & Rogers, M. P. (2011). *Smart smartphone development: iOS versus android*. Paper presented at the Proceedings of the 42nd ACM technical symposium on Computer science education, Dallas, TX, USA.

[13] Goldman, M., & Miller, R. C. (2009). Codetrail: Connecting source code and web resources. *J. Vis. Lang. Comput., 20*(4), 223-235. doi: 10.1016/j.jvlc.2009.04.003

[14] Hartmann, B., Dhillon, M., & Chan, M. K. (2011). *HyperSource: bridging the gap between source and code-related web sites*. Paper presented at the Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, Vancouver, BC, Canada.

[15] Hartmann, B., Doorley, S., & Klemmer, S. R. (2008). Hacking, Mashing, Gluing: Understanding Opportunistic Design. *IEEE Pervasive Computing, 7*(3), 46-54. doi: 10.1109/mprv.2008.54

[16] Hartmann, B., MacDougall, D., Brandt, J., & Klemmer, S. R. (2010). *What would other programmers do: suggesting solutions to error messages*. Paper presented at the Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, Atlanta, Georgia, USA.

[17] Hoffmann, R., Fogarty, J., & Weld, D. S. (2007). *Assieme: finding and leveraging implicit references in a web search interface for*

programmers. Paper presented at the Proceedings of the 20th annual ACM symposium on User interface software and technology, Newport, Rhode Island, USA.

[18] Hou, D., & Li, L. (2011). *Obstacles in Using Frameworks and APIs: An Exploratory Study of Programmers' Newsgroup Discussions*. Paper presented at the Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension.

[19] Hundhausen, C. D., Brown, J. L., Farley, S., & Skarpas, D. (2006). *A methodology for analyzing the temporal evolution of novice programs based on semantic components*. Paper presented at the Proceedings of the second international workshop on Computing education research, Canterbury, United Kingdom.

[20] Jablonski, P. (2007). *Managing the copy-and-paste programming practice in modern IDEs*. Paper presented at the Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion, Montreal, Quebec, Canada.

[21] Kim, M., Bergman, L., Lau, T., & Notkin, D. (2004). *An Ethnographic Study of Copy and Paste Programming Practices in OOPL*. Paper presented at the Proceedings of the 2004 International Symposium on Empirical Software Engineering.

[22] Kline, R. B., & Seffah, A. (2005). Evaluation of integrated software development environments: challenges and results from three empirical studies. *Int. J. Hum.-Comput. Stud., 63*(6), 607-627. doi: 10.1016/j.ijhcs.2005.05.002

[23] Ko, A., Myers, B., & Aung, H. (2004). Six learning barriers in end-user programming systems. *Visual Languages and Human-Centric Computing (VL/HCC 2004)*, 199–206.

[24] Mamykina, L., Manoim, B., Mittal, M., Hripcsak, G., & Hartmann, B. (2011). *Design lessons from the fastest q&a site in the west*. Paper presented at the Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, Vancouver, BC, Canada.

[25] O'Brien, M. P., Buckley, J., & Exton, C. (2005). *Empirically Studying Software Practitioners " Bridging the Gap between Theory and Practice*. Paper presented at the Proceedings of the 21st IEEE International Conference on Software Maintenance.

[26] Piech, C., Sahami, M., Koller, D., Cooper, S., & Blikstein, P. (2012). *Modeling how students learn to program*. Paper presented at the Proceedings of the 43rd ACM technical symposium on Computer Science Education, Raleigh, North Carolina, USA.

[27] Riley, D. (2012). *Using mobile phone programming to teach Java and advanced programming to computer scientists*. Paper presented at the Proceedings of the 43rd ACM technical symposium on Computer Science Education, Raleigh, North Carolina, USA.

[28] Serrano, N., Hernantes, J., & Gallardo, G. (2013). Mobile Web Apps. *Software, IEEE, 30*(5), 22-27. doi: 10.1109/MS.2013.111

[29] Sim, S. E., Clarke, C. L. A., & Holt, R. C. (1998). *Archetypal Source Code Searches: A Survey of Software Developers and Maintainers*. Paper presented at the Proceedings of the 6th International Workshop on Program Comprehension.

[30] Storey, M.-A., Damian, D., Michaud, J., Myers, D., Mindel, M., German, D., . . . Hargreaves, E. (2003). *Improving the usability of Eclipse for novice programmers*. Paper presented at the Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange, Anaheim, California.

[31] Stylos, J., & Myers, B. A. (2006). *Mica: A Web-Search Tool for Finding API Components and Examples*. Paper presented at the Proceedings of the Visual Languages and Human-Centric Computing.

[32] Tillmann, N., Moskal, M., Halleux, J. d., Fahndrich, M., Bishop, J., Samuel, A., & Xie, T. (2012). *The future of teaching programming is on mobile devices*. Paper presented at the Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education, Haifa, Israel.

[33] Treude, C., Barzilay, O., & Storey, M.-A. (2011). *How do programmers ask and answer questions on the web? (NIER track)*. Paper presented at the Proceedings of the 33rd International Conference on Software Engineering, Waikiki, Honolulu, HI, USA.

[34] Wasserman, A. I. (2010). *Software engineering issues for mobile application development*. Paper presented at the Proceedings of the FSE/SDP workshop on Future of software engineering research, Santa Fe, New Mexico, USA.

[35] Wu, H., Guo, Y., & Seaman, C. B. (2009). *Analyzing video data: A study of programming behavior under two software engineering paradigms*. Paper presented at the Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement.

[36] Yeh, R. B., Paepcke, A., & Klemmer, S. R. (2008). *Iterative design and evaluation of an event architecture for pen-and-paper interfaces*. Paper presented at the Proceedings of the 21st annual ACM symposium on User interface software and technology, Monterey, CA, USA.

[37] Yoon, Y., & Myers, B. A. (2011). *Capturing and analyzing low-level events from the code editor*. Paper presented at the Proceedings of the 3rd ACM SIGPLAN workshop on Evaluation and usability of programming languages and tools, Portland, Oregon, USA.

[38]