Characterizing Developer Behavior in Cloud Based IDEs

Yi Wang[1], Patrick Wagstrom[2], Evelyn Duesterwald[2], David Redmiles[1]
[1] Department of Informatics, University of California, Irvine, CA 92697
[2] IBM T. J. Watson Research Center, Yorktown Heights, NY 10598
[1] {yiw, redmiles}@ics.uci.edu
[2] {pwagstro, duester}@us.ibm.com

ABSTRACT
Cloud based integrated development environments (IDEs) provide many advantages for deployment and maintenance of programmer environments. In addition to running on a large number of end user devices via a web browser, they require no local installation or upgrading. They also, by virtue of the fact that the system is a primarily hosted on a remote server, allow detailed observation of developer behaviors. In this paper we present details of a first-of-a-kind study of user behavior in a cloud based IDE, JazzHub. We utilize the features of JazzHub to monitor the growth of files and correlate this data with activity observed on the screen to build a state transition model for software developers that includes common development tasks, such as CODING and DEBUGGING, but also captures extended interactions with remote resources. These findings are an early step toward realizing the potential for enhanced interactions in cloud based IDEs.

Categories and Subject Descriptors
D.2.6 [**Software Engineering**]: Programming Environments

General Terms
Design, Human Factors, Measurement

Keywords
Cloud based IDE, developer behavior, JazzHub, user study

1. INTRODUCTION
Cloud based computing has rapidly evolved to a mainstream component of business environments. According to the 2013 Eclipse Community Survey, 53% developers have developed or deployed cloud based applications, more than doubling the 2012 rate of only 21%[1]. However, to this point much of the attention on cloud based applications has focused on deployment, execution, and management of applications - sometimes collectively called devops. With nearly ubiquitous access to the internet, and the variety of devices, laptops, tablets, and phones, that can support rich applications, we can assert that the next mainstream software development environments will be cloud based IDEs, such as Cloud9, FriendCode and JazzHub [14].

While the percentage of users adopting cloud based IDEs is significantly less than the percentage of users developing cloud based services – in 2013 only 13.3% of developers had adopted cloud based IDEs – the adoption rate increased 300% between 2012 and 2013 [1]. These cloud based IDEs provide a number of advantages. They can take advantage of the rich opportunities afforded by the client-server model of web applications. These advantages include for example, automatic saving and backup of code, real time pair programming across a distance, and the ability to easily harness a huge number of remote computers to perform tasks, such as build and search, that previously were done on the desktop. These IDEs make the development process truly scalable and flexible: no installation on the desktop is required, potential collaborations among unlimited numbers of developers is enabled, and maintenance is simplified as new versions can be automatically provisioned to all users.

Cloud based IDEs also provide another great advantage for developers, managers, tool builders, and researchers alike: they provide a centralized point for the transparent collection of data related to the software development process. Thanks to improvements in JavaScript and underlying communications protocols such as WebSockets – which allow for every keystroke on a web page to be easily sent back to a server, and WebRTC, which provides a mechanism to stream video and audio directly from the browser to the server – it's possible to see, analyze, and perhaps even understand every action that users take while developing software. This unprecedented transparency enables us to extract insights about the behavior of the community of IDE users that can be exploited in many new and novel ways. Developers may be provided with relevant and real-time assistance based on not only their own behavior but also based on leveraging the experience of the entire developer population. Development managers may gain new insights about individual and team contribution and finally, the IDE may leverage real-time insights about actual tool and feature usage and user experience to improve future versions of the IDEs.

To take full advantage of these opportunities, we must have an accurate understanding of how users behave when they develop programs in cloud based IDEs. This is important for a number of reasons. First, studying user behavior allows us to evaluate existing cloud based IDEs and can provide insights for better designs and features [18]. Second,

accurate models of user behavior may be applied in developing next-generation recommendation/assistance systems that requires the understandings of user activity to build work context [20]. Third, an in depth understanding of evolving trends in user software development behavior is broadly important to the wider research field of software engineering [9].

In this paper, we report a first-of-a-kind in-depth analysis of developer behaviors of using cloud based IDEs to solve small-scale programming problems. Using the JazzHub[1] environment from IBM as the main platform, we conducted a laboratory user study to explore how developers behave in using cloud based IDEs to solve small programming tasks. We collected data from different sources, including video recording of the participant's screen, captures of all HTTP and HTTPS sessions between the participant any and remote server. We also requested explicit user feedback in form of a questionnaire, and we collected all artifacts produced by each participant during the study. We provide new insights into developer behavior in cloud based IDEs through quantitative and qualitative analyses of the collected data.

Specifically, the contributions of this study are:

- A comprehensive list of activities performed by developers in using JazzHub, and their frequency and time distributions.
- A first order Markov model that describes the activity transition from one to another. The model captures the dominated transition patterns among different activities. The analyses of activity transitions lead to new insights about developers' behavior in Cloud based IDEs.
- A method to classify code growth trajectories. By combining this information with the observed activities, we developed a refined and more accurate activity transition model for a select activity: the CHANGE activity (merged from two activities: CODING and DEBUGGING).

Our study provides a first look into fine-grained developer behavior in cloud based IDEs. We believe that this study and its findings have implications for future improvement of cloud based IDEs, and the development of assistance tools and software analytics based on actual user behavior models.

The rest of this paper is organized as follows. Section 2 briefly summarizes some related work. Section 3 introduces the study design and execution. Section 4 describes the collected data and how the data was prepared and analyzed. Section 5 presents the results while Section 6 discusses the implications, limitations, and other related issues. Finally Section 7 presents a summary of our overall findings and conclusions.

2. RELATED WORK

Empirical research studies on developer behaviors while programming dates back to the late 1970s. In an early paper, Brooks discussed how to experimentally study computer programming [5]. Other early work includes Basili et al.'s studies on Ada development in the 1980's that used self-reported data collected through questionnaires [2]. Although there are still many studies of exploring developer behaviors, most of them are not stand-alone work but often appear in the evaluation sections of specific software tools [9], or as a part of studying a different problem, e.g., knowledge of the code [10], programming comprehension [13], software reuse [21], end-user programming [4, 15], etc. Although these studies provided rich evidences about how developer performing programming tasks, they did not provide systematic, detailed investigation that is comparable to what we did in this paper. Roehm and Maalej [20] provided a conceptual method using Hidden Markov Model to predict programming activity with observed data of interacting with IDE. However, their study only used one subject's data (also one of the authors), which makes their results are much weaker and less valid than us. Besides, they also did not develop the real descriptive model of developers' activity.

Some existing work already utilized the data generated from programming process to develop insights into developers behaviors and task characteristics. A record of actual user interactions can be used to create a "work context" of the user's behavior. These work contexts enable tool designers to provide more relevant support by providing the information users need at their fingertips. Early work in this area by Murphy et al. investigated how different Eclipse features were used by Java developers [18]. Their work was extended from monitoring feature usage to constructing task contexts for making recommendations [11, 12] and identifying developer attributes [10]. Other work developed metrics from monitoring programming activities in order to predict software defects [16]. The results of these studies are mostly simple tool usage statistics which may not well reflect developers' actually behavior. Besides, extracting information from HTTP conversations (the way we used to collect code growth data) is potentially much more scalable than installing plug-ins to developers' desktop with the popularity of cloud based IDEs. A simple passive internet traffic monitor on server-side is capable to capture rich interaction data of thousands of users. We view this as an opportunity of develop better software analytics and enhanced insights. We introduced this concept in our recent work [24].

There are also some work using activity transition to evaluate software tools. For example, Mangano and van der Hoek used activity transition diagrams to evaluate the influence of a whiteboard design tool on the process of software design [17]. The study described in this paper serves for different goal. We focused more on characterizing developers' behaviors rather than evaluating how new programming environments (JazzHub) changed their behaviors. The behavioral models developed in this paper are more comprehensive and detailed as we not only enumerate the transitions but also estimate their probabilities of these transitions. However, it would be very useful to explore the changes of people's behaviors resulting from the adoption of new cloud based IDEs in future study.

3. STUDY DESIGN AND EXECUTION

In this section, we introduce the design and execution of a laboratory user study in which we collected empirical data as described in the next section. The study design followed

[1]JazzHub: https://hub.jazz.net/

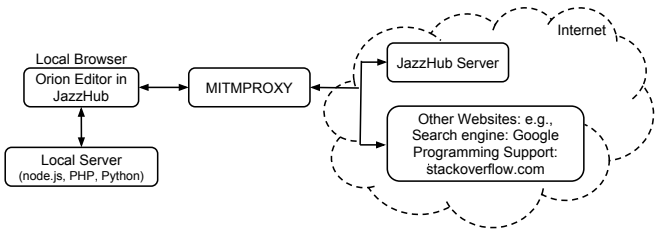

Figure 1: The user study environment.

the guidelines proposed by Wohlin [26]. We also consulted external experts in human subject design processes to evaluate and ensure that, to the maximal degree possible, our results would be unbiased.

3.1 Building User Study Environment

We extended and enhanced a cloud based IDE to provide additional functionality necessary for a study. The main platform for programming in our study is the Orion editor integrated in JazzHub. Subjects were asked to use Orion to perform a set of programming tasks. Three languages (Javascript, PHP, and Python) were provided for subjects to choose. We enabled the "autosave" function of Orion editor, so that the server copy of source code files were periodically uploaded to the server with the contents of the local web browser. Hence, we can analyze how source code files change, both in terms of size and content, by examining these HTTP conversations. As JazzHub was still under heavy development at the time of our study, some elements had to be simulated for our study. In particular deployment was simulated using a browser extension that ran the code present in the editor on the user's local machine and reported the result back to the browser window. MITMPROXY[2] was used to capture all users' internet traffic which include the interaction with the JazzHub server and other websites (e.g., Google, Stackoverflow, etc). The architecture of our experiment environment is shown in figure 1.

A dedicated JazzHub account was created for this study and was used by all users in the study. We created a set of projects which contain simple source code skeleton files for each task using this account. This eliminated a potentially confounding factor by not requiring subjects to create their own projects and source code files. It also eliminated the need for subjects to learn and deal with IDE project set up and management so subjects could focus on implementing the functions required for each programming task. In a user study with a time limit, this greatly facilitated the collection of more data relevant to actual programming behavior.

3.2 Subjects

Before the execution of the user study, we conducted a pilot study with a summer research intern at the IBM T. J. Watson Research Center to test the study design and study environment. Slight modifications to the initial protocol were made - primarily to make the tasks easier - and eight study subjects were enrolled through face to face contact and email invitations. Seven subjects were summer research interns who were also enrolled in various research Ph.D. programs during the study and one subject was a new full time IBM research staff member. Before the study, we communicated with each subject to learn their programming language preferences for the user study environment. Surprisingly, all subjects chose `Python` although we offered three alternatives. The subjects were given a $10 lunch coupon as the compensation for participating this user study. Before the study, we announced that the top 20% performers would be given an additional lunch coupon.

[2]http://mitmproxy.org/

3.3 User Study Tasks

Nine simple programming puzzles were developed for the subjects to solve in the user study. These included one warm-up task and eight programming tasks. The warm-up task (printing "Hello World!" 100 times) is very simple and was not evaluated. The remaining eight tasks (see APPENDIX for details) constituted the main part of the user study. We obtained the tasks from a variety of online sources of programming tasks and technical interview questions common in interviews for software developers.

We used a series of small tasks instead of a single more complex task for following reasons: (1) We wanted to maximize the amount of time that the subjects spent in the JazzHub code editor. Complex tasks not only take longer, but require more time designing a solution outside of the JazzHub code editor. (2) Although these tasks are very small (all can be implemented in less than 40 lines of code), they were carefully selected to cover the fundamentals of a variety of programming concepts, thereby allowing for broader coverage than we could have achieved with a single more complex problem. (3) The use of small tasks allowed us to capture more tasks and traces within the given time constraints. The tasks were ordered from easy to difficult. This allowed subjects to quickly gain a sense of accomplishment from finishing the first tasks which hopefully motivated the sustained application of effort into the following more difficult tasks.

3.4 Study Process

All user study sessions were conducted in August and September 2013. A single user study session contains three main parts. Most subjects finished the user study in 90 minutes. Only one subject participated in each user study session (i.e. no parallel user study sessions were held). The user study was performed in this way to avoid the interference among different subjects. At least two experimenters were present for each user study session. Figure 2 shows the whole process of the study. Two short breaks were arranged during each user study session. Light refreshments were served during the breaks (water, soda, and snacks). The subjects were free to ask for additional breaks at any time point. All eight user study sessions followed an identical protocol and were conducted in the same room with the same hardware.

3.4.1 Part 1: Study Briefing and Warm-up

The experimenters first briefly introduced the study, and asked the subject to sign a experimental consent form that informed the subject of their rights and described the safeguards to ensure their data remained confidential. This was followed by a training session consisting of two steps: In the first step, the experimenter explained how to use JazzHub to develop and run programs. This part usually took around 10 minutes. Then, the subject had up to 15 minutes to get

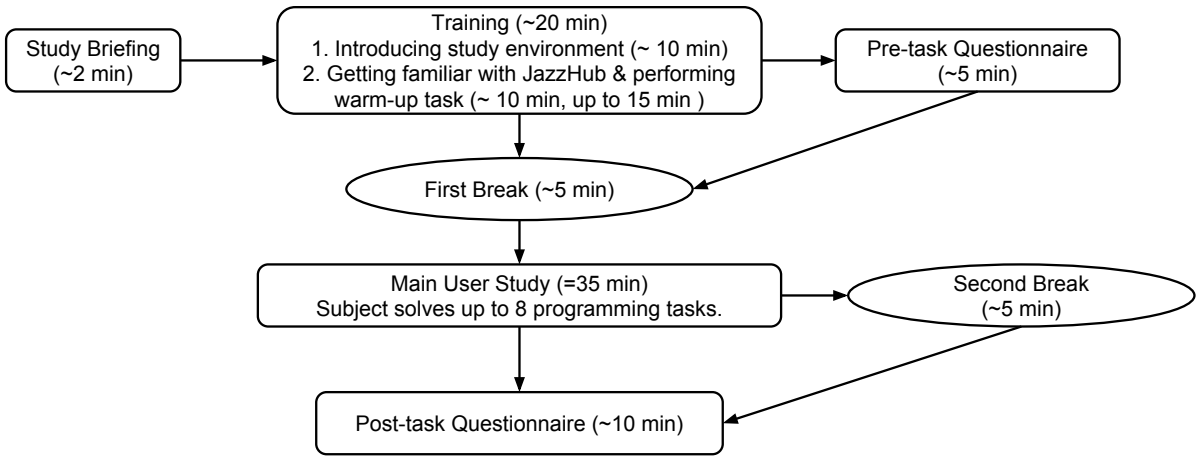

Figure 2: The user study process.

familiar with the study environment and solve the warm-up task. After the subject signaled he or she finished this part, the experimenter checked the solution to ensure the warm-up task was properly finished. Then, the subject was asked to finish a pre-task questionnaire that was designed to capture their initial impression about JazzHub and their self-efficacy about programming using the programming language of their choice. The first break was arranged after this part to prepare the subjects for the more mentally intensive main part of the user study.

3.4.2 Part 2: Main User Study

The subjects were asked to solve as many programming tasks as possible. A new task would be given to the subjects only after they successfully completed a task. After each task, the subjects were asked to fill out a very short 6-question NASA Task Load Index (TLX)[3] survey to assess his workload as the reference for data analysis. The second break was arranged after this part. During the break, the experimenter saved the data and collected all other materials.

The subjects were instructed that their primary goal should be solving as many programming tasks as possible in the allocated time (35 minutes). Improvement on time and memory efficiency of their solutions was presented as secondary goal contributing to their overall performance evaluation. The user study was conducted with a MacBook Pro Laptop with internet connection. The subjects had the freedom to use any online resources to assist in their programming tasks. Pen and paper were provided for potential use. The subject could also use the whiteboard in the study room.

Experimenters collected all papers notes created by the subject and took pictures of the whiteboard. A recording of the subject's computer screen history and an audio recording were also created, but no video was taken of the room or subject. MITMPROXY captured the communications between the computer used in the study and remote servers, which were not restricted to JazzHub, but also included all websites the subject visited during the study session. Therefore, we explicitly required the subject not to login to any website that requires personal "*username*" or "*password*". This does not measurably influence the use of web resources to support programming, for most online programming resources do not require access credentials to be provided.

[3]http://humansystems.arc.nasa.gov/groups/tlx/

3.4.3 Part 3: Post-task Questionnaire Survey

After the second break, the subjects answered the post-task questionnaire survey, which took around 10 minutes on average. The subjects provided their experience and overall assessment of using the Orion editor in JazzHub to solve programming problems. Some basic demographic information was also collected in this questionnaire that was later associated with their experimental data.

4. DATASETS AND ANALYSIS

4.1 Data Description

The user study generated four types of data. The first type is the screen history and audio information collected during the user study. In total, we have 362 minutes and 57 seconds of screen history data (with synchronized audio). The screen history data contains comprehensive information about the subjects' interactions with JazzHub and other websites. This dataset enables us to precisely identify developer's behaviors in using JazzHub.

The second type are the HTTP conversations for traffic exchange through MITMPROXY. The dataset contains 2750 HTTP requests and include all standard HTTP information. As described earlier, we enabled the JazzHub autosave function in the Orion source code editor. With this setting the user's browser periodically uses the `PUT` method to update the server-side copy of the current source file with the contents of the editor window in the local web browser (approximately every 20 seconds during active editing of the file). In total, there are 725 (26.36%) HTTP requests for server-copy source code file updates. Other types of HTTP requests include requests for search engines, online programming resources, etc.

The third type are the artifacts produced by the subjects in the user study sessions. The source code they developed was saved, as well as their hand writings and drawings during

the coding process. The last set of data comes from the questionnaires. In this study, we mostly utilized the first two datasets to build behavior activity model in our data analyses. The third and fourth datasets are used as ancillary resources.

4.2 Data Analysis

The study was configured to collect a myriad of different pieces of data. For this paper we primarily focus on the patterns of the growth of the code and qualitative coding of screen histories. This section describes the capture and transformation processes we employed to make use of the data.

4.2.1 Extraction of Code Growth Data

As previously stated, we used MITMPROXY to capture all HTTP and HTTPS during the study. When a request is captured by MITMPROXY it is tagged with additional metadata including the exact time and destination of the request. We filtered this data for `PUT` requests to the JazzHub server and, as each request contained a copy of the editor at the time, were able to calculate the size of the current file at each point in time in a method that would be amenable to implementation on a cloud based system. We extracted this information for each task and represented the task's source code growth traces as a series of tuples $< time, length >$. In total, 41 traces were extracted. The traces for different subjects vary in length depending on the amount of auto-save events and overall time spent on a task.

As an example, Figure 3 depicts a typical code growth trace. Please note that, the source code file length is usually not zero at the beginning since we provided skeleton solutions for each task. We normalized the source code file length trace to make them comparable across tasks and different starting file length values. Currently, JazzHub does not allow us to specify a fixed time interval between `PUT` invocations in autosave mode, so sample points are not evenly distributed over time. In general, the code growth curve trends upwards, but also exhibits downward when a subject deletes previously written code. The specific shape of a code growth trace can vary significantly between subjects and from task to task, coinciding with variations in programming styles which have been observed by others [8]. Our prior analyses indicate that the shape of a code growth trace is more unique to an individual developer than it is to an individual task [24].

4.2.2 Identifying Code Growth Trajectories

The autosave sample points (i.e., every `PUT` request to update the server copy) divide a code growth trace into a series of segments. Code growth iusually varies across subsequent segments (see figure 3). In some segments, the source code file grows very quick while in others it grows much more gradually or it even decreases. We define the code growth speed v to be the rate of change in source code file size across sampling segments: $v = \Delta S/\Delta T$, where S is the source code size and T is Time. Using code growth speed, we define five trajectories of code growth to characterize the code growth trajectory of each segment: (*Quick Growth*, *Slow Growth*, *Flat*, *Slow Decrease*, and *Quick Decrease*) The method for determining the type of a specific segment contains three steps:

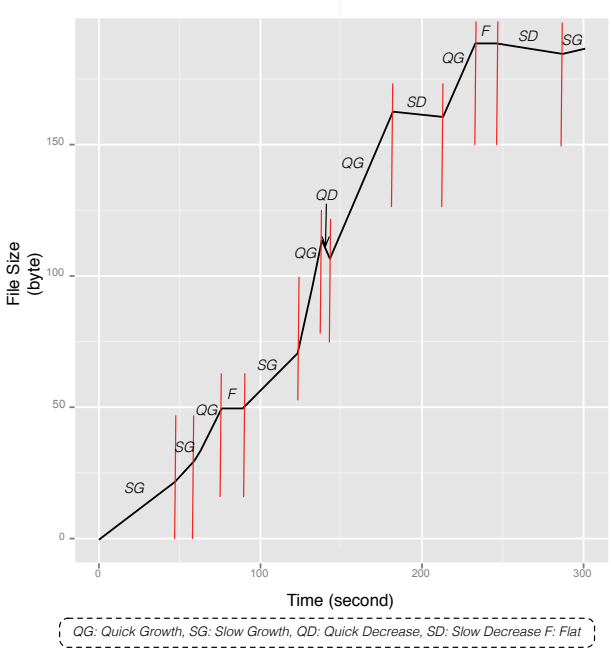

Figure 3: An example of code growth trace (subject no.: 6, task no.: 6). All segments' growth trajectories are marked.

`Suppose there are` n `segments in a code growth trace, and` $i \leq n$`:`
Step 1: `Calculate the average code growth speed (`$\bar{v}$`) for the code growth trace;`
Step 2: `Calculate code growth speed (`v_i`) for segment` i`;`
Step 3:

```
case v_i > v̄:   Quick Growth;
case 0 < v_i ≤ v̄:   Slow Growth;
case v_i = 0:   Flat;
case 0 > v_i ≥ −v̄:   Slow Decrease;
case v_i < −v̄:   Quick Decrease.
```

`Repeat step 2 and 3 to label all segments.`

The method listed above acknowledges the fact that there are individual differences among developers regarding their programming habits and skill levels [24]. Some individual developers may write code very quickly while others code more slowly. Thus, code growth trajectories are unique to each developer and there doesn't exist unified standard regardless of individual differences. In figure 3, we marked each segment's growth trajectory. There are 4 segments of *Quick Growth* segments, 4 of *Slow Growth* , 2 of *Flat*, 1 of *Quick Decrease*, and 2 of *Slow Decrease*. Of course, not all code growth traces exhibit segments of all five trajectories.

4.2.3 Screen History Coding

We followed standard qualitative data analysis methods [7] and best practices from the software engineering literature [23, 25] to code the screen history data. The unit of coding is the "activity" which is defined as a subject's single integrated, meaningful behavior during a programming task. An activity may contain a set of low-level actions, such as moving the mouse or text input. However, these actions are not individually coded as they are too fine-grained to help in

Activity	Activity Description	# Instance (%)	# Time (%)
Navigation	The subject navigates to find the directory or pages they need.	51 (4.49%)	10m54s (4.35%)
Reading Questions	The subject read and understand task at the beginning of each task.	36 (3.17%)	14m35s (5.82%)
Searching	The subject uses search engine to find online resources to support his problem solving.	59 (5.19%)	9m12s (3.67%)
Reading Search Results	The subject reads the search results to decide which links to click.	56 (4.93%)	5m59s (2.39%)
Processing Search Results	The subject decides how to deal with the search results. It is different from Reading Search Result as the subject decides not to visit any of the links.	12 (1.06%)	2m29s (0.99%)
Viewing Web Resources	The subject views online programming support resources.	115 (10.12%)	24m26s (9.74%)
Coding	The subject works on source code in Orion editor.	303 (26.67%)	105m44s (42.16%)
Run	The subject uses the simulated cloud environment to run the program and view the results.	245 (21.57%)	29m53s (11.92%)
Debugging	The subject tries to fix the problem in the code.	91 (8.01%)	18m02s (7.19%)
Accidents	The subject randomly switches between different pages without clear reason.	29 (2.55%)	2m46s (1.10%)
Idle	The subject does nothing for a specific period ($\geq 3seconds$).	139 (12.24%)	26m47s (10.68%)

Table 1: Enumeration of all activities and their occurrences.(Total Instances: 1,136, Total Time: 250m47s)

the understanding of programmer behavior. The analysis level was similarly adopted in prior studies [20, 22]. Audio data was used when video data would not provide sufficient cues to determine an activity. The qualitative data was coded following the open coding protocol. Codes for activities emerged from the coding process. We maintained a master list of codes (codebook) in the coding process. An external researcher with extensive qualitative research experiences (over 30 years) was invited as the referee to judge the validity of the codes. Each activity was coded in the following standard form:

```
Start Time, End Time, Activity (Attributes) #Note
```

As the names indicate, "`Start Time`" and "`End Time`" specify the timestamp and duration of the activity. The activity is associated with attributes (if necessary) to provide additional detail. In our codebook, we specified a set of options for different attributes of an activity. If additional clarifications were need, we put them in the "Note" section starting with a "#". For example, if we observe that a Coding activity includes copying code from stackoverflow, we would record that information as a note. Here is an example from our dataset:

```
09:31, 09:33, Searching (Search Term: "python module", Action: "Search Box" in "Current tab")
```

In this example, Searching is the central activity. The searching activity took 3 seconds. The attributes provide more details about this activity, such as the search term ("Search Term" attribute) and how developers performed this search ("Action" attribute). This example does not contain a note. The main benefit of using semi-structured codes is to enable some level of automatic data analysis (e.g., parsing). These codes may also enable future explorations beyond characterizing user behaviors, such as activity visualization and mining temporal patterns.

4.3 Data Limitations

Our datasets provided us with a unique opportunity to study user activities in using a cloud based IDE to develop programs through a combination of qualitative data coding and code growth data extracted from internet traffic data. However, the dataset also comes with some limitations. First, the sample is relative small. We only have 8 subjects and 41 finished tasks – although as noted in our previous research, the subjects had a diversity of skill levels [24]. Second, the programming tasks were small exercises rather than real-word software development tasks. Therefore, our findings about developers' activities may change when JazzHub is used to develop larger applications. Third, all subjects chose to use `Python`, so the results may be less valid for another language. However, by carefully selecting study tasks, we minimized the influences of specific language features and focused the subjects on programming fundamentals (e.g. string operations). Lastly, we use JazzHub as the platform for this study, hence, the results may be different for other cloud based IDEs. However, our study focused primarily on the editing function in the Orion editor which is fairly representative of current cloud based code editors. Replication on other cloud based IDEs will help to enhance the confidence in our results or to identify the limitation of our findings [3]. Some further discussions are provided in section 6.3.

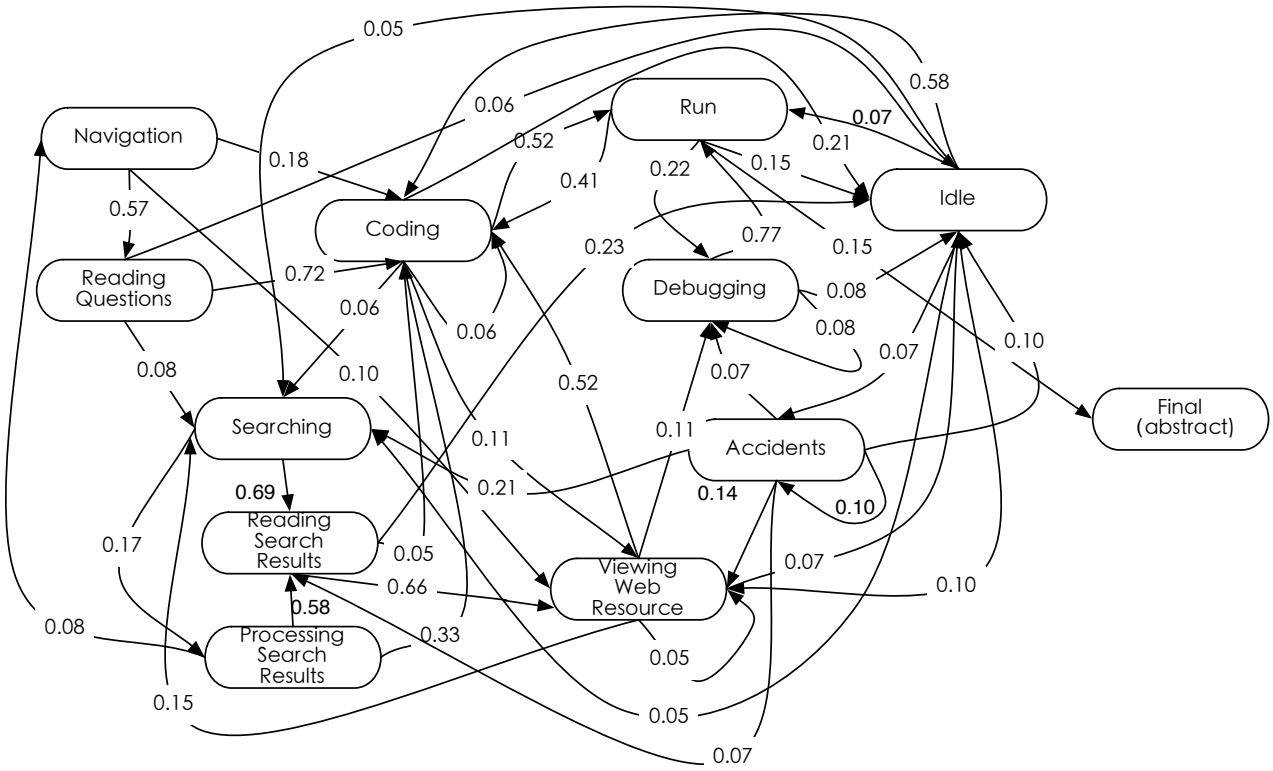

Figure 4: Transition probability of user behaviors in using JazzHub code editor (at the level of activities). Note: Edges with probability less than 5% were removed.

5. RESULTS

5.1 User Activities in JazzHub

During a first pass of characterizing user behavior, we identified 11 unique activities that have at least one occurrence in the coded video and audio data. Table 1 displays these activities and their descriptions with the number and share of each activity's instances, and the elapsed time taken up by these instances. We observe that CODING and RUN are two most frequent activities. However, we note that there was no single activity with a majority share of the instances or a majority share of elapsed time. CODING accounts for only 26.67% of instances and 42.16% of elapsed time. Furthermore, the cumulative time share of all activities related to the interaction with JazzHub (i.e., NAVIGATION, CODING, RUN, DEBUGGING) barely exceeds 65% of the total subject observation time. Thus, over 30% of the time was spent on activities outside of JazzHub. Such a high proportion of time spent outside the development environment may suggest new opportunities to improve developers' productivity by integrating theses additional activities into the environment.

CODING is also the activity with the longest average duration (20.94s) while ACCIDENTS has the shortest duration(5.72s). The shorter duration ACCIDENTS (e.g., open a new tab in browser, then close it) activity suggests that the developers quickly recover from confusions. An activity similar to CODING is DEBUGGING in that both of them most of the time refer to changing source code. In our coding standard, only those changes to code that are explicitly associated with a "bug" (unexpected results of RUN) are coded as DEBUGGING. In general, a single CODING activity is much longer than a single DEBUGGING activity (20.94s vs. 11.89s), indicating the changes to source code were usually smaller in DEBUGGING.

5.2 Activity Transition

During the second step of characterizing user behavior, we constructed a first-order Markov chain of user activity based on the sequence of activities seen in all sessions. We added one abstract activity[4], FINAL, which we appended to the sequence of activities at the end of the programming tasks. Figure 4 shows the transitions of user activities. The directed edges represent the transition from one activity to another, and the numbers on them indicate the transition probability. Formally, the transition probability can be defined as:

$$P(Activity_t = i | Activity_{t-1} = j) \quad (1)$$

Edges with probability less than 5% were removed to simplify the diagram. Therefore, the sum of all outgoing probabilities (excluding the omitted edges) for each activity may be slightly lower than 1.0. This diagram provides a holistic view of developers' behavior.

The diagram allows for a few interesting observations. First of all, activity sequences usually start at NAVIGATION followed by READING QUESTIONS. Naturally, developers usually spend some time understanding the task. However, there are still 18% of NAVIGATION activities that directly lead to

[4]We did not create an abstract activity for START as almost all sequences (40 in 41) start from NAVIGATION.

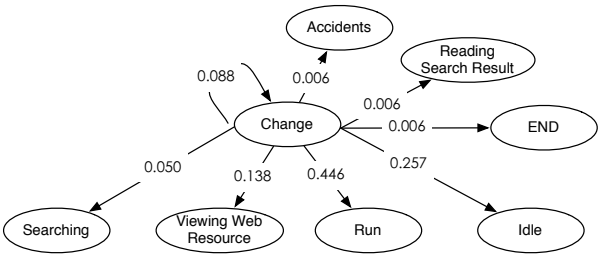

Figure 5-1. Activity transition (from ``Change'') under **Quick Growth** condition (n = 159).

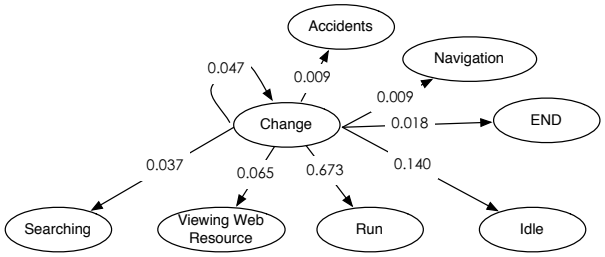

Figure 5-2. Activity transition (from ``Change'') under **Slow Growth** condition (n = 107).

Figure 5-3. Activity transition (from ``Change'') under **Quick Decrease** condition (n = 21).

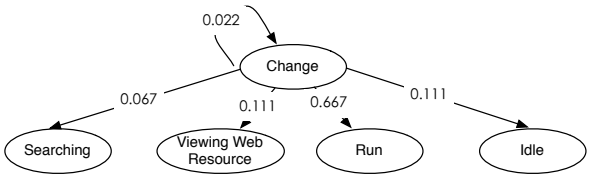

Figure 5-4. Activity transition (from ``Change'') under **Slow Decrease** condition (n = 45).

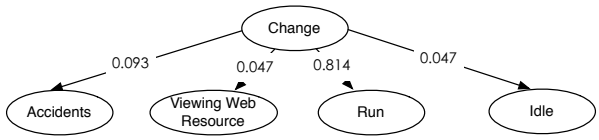

Figure 5-5. Activity transition (from ``Change'') under **Flat** condition (n = 43).

Figure 5: Transition probability from Change to other activities in different code code growth trajectories.

Coding, indicating that some individuals may have followed a "code first, think later" strategy in their problem solving. We observe that there are two prominent loops of activities which are Coding-Run, and Debugging-Run. These loops show that iteratively problem solving is very common even for small problems. However, self loops are very rare. The biggest transition probability of self loop is only 10% for the activity Accidents, which only has 29 instances.

Another interesting observation is that it appears that the subjects don't seem to have a very clear strategy on how to search the relevant resources even for very simple tasks of our study. In our sample, the transition probability from Viewing Web Resource to Searching is 17%. This means over 1/6 web resources from the initial searches apparently did not satisfy the information needs of the subjects. A closer inspection showed that most of the unsatisfactory searches resulted from using improper search terms. The challenge of finding relevant documents could be partially remedied by an intelligent tool that provides programming and task specific instant search. While such a feature is certainly possible in a traditional IDE, a cloud based IDE can easily and quickly learn from a large population of user searches and tasks, leading to a much more effective experience [24].

5.3 Activity Transition and Code Growth Trajectories

The third step of characterizing user behavior combines the code growth data and the qualitative coding of activities. In section 4.2.2, we showed that a single code growth trace can be viewed as a sequence of segments of different trajectories. In general, an instance of an activity is slightly shorter than an instance of a code growth segment ($\approx$ 15 seconds vs. $\approx$ 20 seconds). Thus, we were able to map each activity, or at least its main part, to the matching containing segment by synchronizing activity and segment breakdowns. This allowed the creation of a mapping between the code growth trajectory of the matched segment and the activity.

We are specifically interested in the mappings with the Coding and Debugging as these are activities that we can infer without the screen history data using only code growth data. Screen history data is usually not available to a cloud based IDE provider, but code growth data could easily be tracked by the IDE server. If code growth or code decrease is detected during a time interval, it must coincide with at lease one Coding or Debugging activity in this time interval[5] It is possible that different growth trajectories correlate with different activity transition probabilities from Coding or Debugging to other activities. To explore this, we first merged these two activities into a single super-set activity and labeled it the Change activity because looking at only code growth data we cannot tell the difference between Coding and Debugging. Then, we calculate the conditional activity transfer probability, which can be formally defined as:

$$P(Activity_t = i | Activity_{t-1} = \text{“}Change\text{”}, \\ if\ (code\ growth\ trajectory_{t-1} = m), \\ where\ m \in \{QG, SG, F, QD, SD\} \quad (2)$$

Figure 5 shows the activity transitions for five different trajectory trajectories. We kept all edges in these diagrams. Please note that there are in total 375 instances (out of 395)

[5]Note that these two activities also could happen even during a *Flat* code growth trajectory segment.

of CHANGE activities. 20 instances of CHANGE (15 of CODING, 5 of DEBUGGING) were not paired with a code growth trajectory. The reason is that there are 12 CODING activities occurring at the very beginning of the corresponding traces, so that they were not captured by HTTP conversations. For the remaining 8 activities we do not having coinciding code growth data due to a server connection loss that occurred when subject 4 was performing her third task, which led to the loss of 3 CODING and 5 DEBUGGING activities.

Unsurprisingly, the most likely next activity after a CHANGE is a RUN activity for all five trajectories. However, the transition probabilities vary dramatically, ranging from 44.6% in the *Quick Growth* trajectory to 81.9% in *Quick Decrease*. This probability under *Slow Growth* is 23% more than that under *Quick Growth*. The reason is that the *Slow Growth* trajectory usually only occurred after the subjects had already finished the main body of the program and were working on small details or debugging. These smaller-scale cleanup activities usually lead to RUN in order to examine and validate the outputs. In contrast, based on our observations, a *Quick Growth* trajectory usually implied that developers were still working on the main body of the program.

Another noteworthy aspect is that the transition probability from CHANGE to VIEWING WEB RESOURCE is twice as high for the *Quick Growth* trajectory compared to the *Slow Growth* trajectory (13.8% vs. 6.5%). This result is somewhat counterintuitive, one might expect that developers rely less on external references when they are in fast coding mode and less likely to experience difficulties. However, the data suggest a different reality. After cross-examining with the screen history video, we found developers often referred to the web resource for quick information about language features (e.g., `remove` method in `Python`) and then returned to coding when they were in *Quick Growth* coding. In the *Slow Growth* coding, developers usually run the program to check the output first after making a small change instead of viewing web resource.

6. DISCUSSION

6.1 Cloud IDE Design Implications

We believe that understanding user activities is important for cloud based IDEs providers and tool designers. Our investigation provides an understanding on how developers use cloud based IDE to solve small programming problems. Despite the increasing popularity of cloud based IDEs, we haven't been able to find a similar detailed study of programming behavior for cloud based environments. In the following, we discuss a number of implications for the design of cloud based IDEs.

A significant advantage provided by cloud based IDEs is that they enable developers to use the development environment from a variety of hardwares such as laptop, tablet, phones. However, the current design of cloud based IDEs restricts the realization of this advantage. Our study reveals that subjects spend 15.79% of development time with searching and viewing related online programming resources, even for the relatively easy programming tasks in our study. When programming on mobile devices with small displays, frequent switches between the programming resources pages and code editor can be very disruptive. In our study, although we used a 15-inch laptop, we still observed several subjects having to adjust the browser size and position to view the resource information next to the code they were working on. Directly integrating web native programming supporting tools (for example, stackoverfolw.com) with the IDEs could free developers from having to frequently switch between different contexts.

Our study shows that 17% of initial searches failed to return satisfying programming resources, so that the developers had to change search terms to start new searches. This caused some productivity loss. As our prior work suggested [6, 24], cloud based IDEs can leverage large amounts of globally collected developer data to provide useful suggestions for a specific developer. One way to improve search terms would be to include a SEARCHING feature in cloud based IDEs. With this feature, the server side of a cloud based IDE can aggregate large amounts of historical searching data, and use these data to provide interactive search query refinement. Functionally, this would exhibit itself as a cross between Intellisense and similar code completion tools and the text autocomplete feature found in most search engines. This is especially valuable to novice developers who have limited experience in quickly finding relevant resources.

The activity transitions we established and their corresponding probabilities may be applied to develop intelligent context-aware programming assistance tools which would require predicting developer activities with a moderate level of accuracy [20]. Consider, for example, a tool with capabilities to automatically switch views in the UI of the IDE based on current, past, and potential future developer activities [11]. The benefits of building a tool for cloud based development is that we can avoid the costly and invasive instrumentation on developers' local computer by passively monitoring the internet traffic. Such as passive monitoring mechanism would be less disruptive to developers' normal behavior.

6.2 Other Potential Applications

The potential applications of characterizing developers behavior in cloud base IDEs extend beyond the design of interactive web applications. It also could be applied to system level tasks such as balancing work load between different servers. For example, given the fact that the subjects may run their program with very high probability under specific code growth trajectories (see figure 4), it is possible to leverage this information to enhance traditional workload balance techniques that automatically switch between different cloud based IDE servers to achieve better workload balance. Furthermore, 7 out of 11 identified activities (e.g., IDLE) do not consume any cloud based IDE server resources. The occurrences and lengths data can be used to fit to proper distribution functions. These best fit functions can be used to generate synthetic (parameterizable) traces, that mimic actual workload for server optimization [19].

6.3 Threats to Validity

Construct Validity. From the perspective of construct validity, we believe there are no significant threats. The construct of a "code growth trace" consists of a series of tuples $< time, length >$. The traces were extracted from automatically generated HTTP requests, and measured without any subjective intervention. Most activity constructs that

emerged from the screen history coding process are straightforward. We carefully examined the differences among similar activities and documented them in our codebook. We also invited an external referee who has over 30 years experience in qualitative research to judge the coding scheme. Hence, there is no significant threat to construct validity.

Internal Validity. From the perspective of internal validity, the rigorous procedures of our study process (see Section 3) and data analysis (see Section 4) ensure that most threats have been overcome. We admit that imposing a time limit was not a good fit for some subjects' preferred programming style and may influenced their behavior.

External Validity. The major challenge of this study's validity comes from the external perspective. In Section 4.3, we already summarized the limitations of the dataset used in this study. Due to limitations of the experimental setting, we cannot guarantee that the results would be still valid in other research or real world setting. However, we are confident that the results are largely applicable to similar settings (small programming tasks, dynamic language) to provide insights about user behavior when programming with cloud based IDEs. In fact, there are a number of real world situations that fit this setting, one example is the Codecademy[6], in which people learn programming through solving small programming problems.

The relative small number of subjects also limits our study's generalizability. When conducting in-depth qualitative analyses of the sort done in this study, practical constraints limit the number of participants, which in turn limits the extent to which findings can be generalized. To reduce this threat, we triangulated our data, analysis methods, and interpretations. For example, the analysis does not purely rely on qualitatively coded data but also objective data extracted from HTTP requests. In this way, we incorporated multiple sources of data, multiple analysis methods, and multiple researchers' perspectives to partially offset the negative effects introduced by small sample size.

7. CONCLUSIONS

In this paper, we reported a detailed study that characterizes developer behavior in cloud based IDEs. To the best of our knowledge, it is a first-of-a-kind analysis of developer behavior in cloud based IDEs. Using data collected from a well designed laboratory user study, we identified developer activities and their occurrences. We also developed a first order Markov model to provide a holistic view of the activity transitions. Using the code growth data, we proposed a refined model to provide more accurate and detailed descriptions of transition from CHANGE activity (the only activity that can be inferred from source code growth data alone) to other activities. The analyses of the descriptive activity transition model suggest new insights into how developers behave in cloud based IDEs. For example, we found they tend to use online programming resources more frequently when they are in *Quick Growth* programming.

Cloud based IDEs are still in their infancy. The findings in our study provide some useful implications on what features may need to be added to cloud based IDEs in the future. We believe studying user behavior will inspire the evolution of cloud based IDE designs and features. In future work, we are planning to continue our efforts on investigating developer interaction with cloud based IDEs with multiple research techniques and settings. This paper presents only a first glimpse into the insights we expect to gain about how developers work in cloud based IDEs. As the community around JazzHub and other cloud based IDEs grows we look forward to continuing our analysis to get a better understanding of how developers work in the real world and how to exploit knowledge of user behavior to provide the next generation of rich and productive environments for software developers.

[6]http://www.codecademy.com/

8. ACKNOWLEDGEMENTS

This work is partially supported by NSF grant 1111446. The authors want to thank all participants of the user study.

9. APPENDIX: USER STUDY TASKS

The following eight user study tasks were used in our study. During the study, we also provided an output example for each task to help subjects better understand the task and to test their programs.

1. **FizzBuzz.**
Write a program that prints the numbers from 1 to 100. But for multiples of three print "Fizz" instead of the number and for the multiples of five print "Buzz". For numbers which are multiples of both three and five print "FizzBuzz".
2. **Power.**
For two non-negative integer x, y, write a program to calculate x^y, without using any type of "`pow()`" or similar build-in methods in standard library.
3. **Move Zero.**
For an array of integers, with some element has value of '0', write a function to move all 0-value elements to the end of the array, while keeping the other elements' relative position unchanged. Your function should return the new array.
4. **Substring.**
For two strings s1 and s2, write a program to check whether or not s2 is a substring of s1. Output "Y" if yes, "N" if not.
5. **Fibonacci Number.**
Write two functions to print out the nth Fibonacci number. The first program should be implemented in **recursion** and the second should be implemented **without recursion**.
6. **Equilibrium Point.**
Given a list of integers, find the *equilibrium point* of the list, output its index and value.
Equilibrium point: The element in a list such that the sum of numbers on the right side of the element is equal to the sum of numbers on the left of the element.
7. **Maximal Sequence.**
Find the **maximal sequence** for an array with both positive and negative integers, and return the **start/end index**. The **maximal sequence** is the subsequence with the largest sum.
8. **Maximal Interval.** *(Any srting to the list is not allowed)* Given a list of integers, find the **maximal interval** in the list, output the start number and end number.
An interval is a sequence of numbers that all numbers in this sequence appear in the given list. The **maximal interval** is the longest interval.

10. REFERENCES

[1] Eclipse community survey (2013). http://wp.me/p1HAa-H9. Accessed: 2013-11-19.

[2] V. Basili, E. Katz, N. Panlilio-Yap, C. Ramsey, and null Shih Chang. Characterization of an ada software development. *Computer*, 18(9):53–65, 1985.

[3] V. Basili, F. Shull, and F. Lanubile. Building knowledge through families of experiments. *Software Engineering, IEEE Transactions on*, 25(4):456–473, Jul 1999.

[4] C. Bogart, M. Burnett, A. Cypher, and C. Scaffidi. End-user programming in the wild: A field study of coscripter scripts. In *Proc. VL/HCC'08*, pages 39–46, 2008.

[5] R. E. Brooks. Studying programmer behavior experimentally: The problems of proper methodology. *Commun. ACM*, 23(4):207–213, Apr. 1980.

[6] M. Bruch, E. Bodden, M. Monperrus, and M. Mezini. IDE 2.0: collective intelligence in software development. In *Proc. FoSER'10*, pages 53–58, 2010.

[7] P. Camic, J. Rhodes, and L. Yardley. *Qualitative Research in Psychology: Expanding Perspectives in Methodology and Design*. American Psychological Association, 2003.

[8] D. Egan. Individual differences in human-computer interaction. In *Handbook of Human-Computer Interaction*. Elsevier, 1988.

[9] S. R. Foster, W. G. Griswold, and S. Lerner. Witchdoctor: Ide support for real-time auto-completion of refactorings. In *Proc. ICSE'12*, pages 222–232, 2012.

[10] T. Fritz, G. C. Murphy, and E. Hill. Does a programmer's activity indicate knowledge of code? In *Proc. ESEC/FSE'07*, pages 341–350, 2007.

[11] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *Proc. FSE'06*, pages 1–11, 2006.

[12] M. Kersten and G. C. Murphy. Task context for knowledge workers. In *Proc. AAAI 2012 Activity Context Representation workshop*, 2012.

[13] J. Koenemann and S. P. Robertson. Expert problem solving strategies for program comprehension. In *Proc. CHI'91*, pages 125–130, 1991.

[14] J. Lautamäki, A. Nieminen, J. Koskinen, et al. Cored: Browser-based collaborative real-time editor for java web applications. In *Proc. CSCW'12*, pages 1307–1316, 2012.

[15] J. Lawrance, C. Bogart, M. Burnett, R. Bellamy, K. Rector, and S. Fleming. How programmers debug, revisited: An information foraging theory perspective. *Software Engineering, IEEE Transactions on*, 39(2):197–215, 2013.

[16] T. Lee, J. Nam, D. Han, S. Kim, and H. P. In. Micro interaction metrics for defect prediction. In *Proc. ESEC/FSE'11*, pages 311–321, 2011.

[17] N. Mangano and A. van der Hoek. The design and evaluation of a tool to support software designers at the whiteboard. *Autom. Softw. Eng.*, 19(4):381–421, 2012.

[18] G. Murphy, M. Kersten, and L. Findlater. How are java software developers using the elipse ide? *Software, IEEE*, 23(4):76–83, 2006.

[19] A. Nazir, S. Raza, and C.-N. Chuah. Unveiling facebook: A measurement study of social network based applications. In *Proc. IMC'08*, pages 43–56, 2008.

[20] T. Roehm and W. Maalej. Automatically detecting developer activities and problems in software development work. In *Proc. ICSE'12*, pages 1261–1264, 2012.

[21] M. Rosson and J. Carroll. Active programming strategies in reuse. In *Proc. ECOOP'93, LNCS 707*, pages 4–20, 1993.

[22] S. Salinger, L. Plonka, and L. Prechelt. A coding scheme development methodology using grounded theory for qualitative analysis of pair programming. *Human Technology*, 4(1):9–25, 2008.

[23] C. Seaman. Qualitative methods in empirical studies of software engineering. *Software Engineering, IEEE Transactions on*, 25(4):557–572, Jul 1999.

[24] Y. Wang, P. Wagstrom, E. Duesterwald, and D. Redmiles. New opportunities for extracting insights from cloud based ides. In *Proc. ICSE'14, NIER Track*, 2014.

[25] C. Wellington and R. Ward. Using video to explore programming thinking among undergraduate students. *J. Comput. Sci. Coll.*, 25(3):149–155, Jan. 2010.

[26] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering*. Computer Science. Springer, 2012.