Comparing Personal Project Metrics to Support Process and Product Improvement

Christoph Aschwanden, Aaron Kagawa
Information and Computer Sciences
University of Hawaii
Honolulu, HI 96822
caschwan@hawaii.edu, kagawaa@hawaii.edu

Abstract

Writing high quality software with a minimum of effort is an important thing to learn. Various personal metric collection processes exist, such as PSP and Hackystat. However, using the personal metric collection processes gives only a partial indication of how a programmer stands amongst his peers. Personal metrics vary greatly amongst programmers and it is not always clear what is the "correct" way to develop software.

This paper compares personal programming characteristics of students in a class environment. Metrics, such as CK Metrics, have been analyzed and compared against a set of similar students in an attempt to find the correct or accepted value for these metrics. It is our belief that programmers can gain much, if not, more information from comparing their personal metrics against other programmers. Preliminary results show that people with more experience in programming produce different metrics than those with less.

1. Introduction

There are many classes taught about software engineering. Many books and papers exist about the subject. However there is no proven way how to design programs. In the mathematics field there are clear guidelines what is allowed or what is not. There are mathematical formulas which can be used to solve problems. The question is, are there similar formulas for software engineering that can be used to define the optimal structure for programs? We use statistical analysis to evaluate software, however results differ.

The Personal Software Process and the Hackystat tool provide great insights into a programmer's personal software development. This paper does not contest that personal metrics helps individual programmers learn about their own process. Rather, we believe that personal metrics are hampered by individualism. Even if programmers are perfectly aware of their own software development process, how will they know if that process is the correct process?

Comparing demographics of users against each other gives a deeper insight on what is going on during the software development process. Comparing graduate students against undergraduate shows how programming behavior changes. It shows how people are learning and are adapting their programming traits.

Our motivation is to compare metrics of different demographics and users against each other. The ability to compare personal metrics will allow the individual to gain insights on how they compare to other programmers. We hypothesize that it allows the individual to gain information on how to improve their own programming. To accomplish the comparisons we have created this general process:

1. Place programmers into groups based on their demographics. Some demographics include: years spent programming, level of education, and the level of classes the programmer has taken.
2. Compare an individual programmer's metrics against other programmers within the same demographic. This allows the programmer to analyze where he stands amongst his peers.
3. Compare the different demographic groups against each other. This allows a programmer to analyze where they stand against other demographic groups.

To accomplish this process we extended the automated metrics collection tool Hackystat. This extension includes creating a questionnaire on the Hackystat server which when filled out, will place a Hackystat user into different demographic groups. We also implemented several analysis charts that provide the user with a comparison analysis of the group which that user is associated with.

If the comparison provides useful information to Hackystat users, then the individual users can make decisions about their own software development process based on their own personal metrics and how they compare to other programmers in Hackystat. It is our belief that these comparisons will become a useful learning tool in addition to personal metrics. This paper provides a pilot study of comparing and analyzing metrics in the collegiate setting, in order to determine its usefulness. Further we analyze different demographic groups to see how programming characteristics are changing. More experienced users are supposed to know better how to program than less experiences ones.

The Related Work section shows how our current effort relates to research done in the past.

2. Related Work

The Personal Software Process (PSP) developed by Watts S. Humphrey was the first to introduce the idea of using software metrics to learn about one's own software development process. PSP helps individual programmers improve their performance by bringing a process discipline to the way to develop software. Using PSP requires the programmer to follow several methods, one of which is the collection and analysis of personal metrics. This method allows programmers to learn about how they program and how they correct their programming process. One of the goals of PSP is, "by measuring their own performance, the engineers can see the effect of these methods on their work" [4]. We have not attempted to use PSP and have not taken the training courses on PSP, however, we believe the individualism that is present in PSP hampers its effectiveness. We believe that there are limitations to the learning that can be answered through personal metrics.

The Hackystat tool [8] created by the Collaborative Software Development Laboratory [7] of the Department of Information and Computer Science at the University of Hawaii at Manoa, is a relatively new tool which automates the collection of personal software metrics. The primary goal of this tool is to automate the collection and analysis of personal software metrics in an attempt to provide its user with features that PSP cannot provide. Like PSP, Hackystat only provides information about a programmer's personal software metrics.

Hackystat will be used in this study to analyze the effectiveness of personal software metrics and will later be explained in further detail.

Object-oriented metrics are software metrics for object-oriented programming languages. Hackystat's analyses and our comparison analyses use a specific set of object-oriented metrics; the Chidamber and Kemerer's OO metrics (CK metrics). CK metrics can be used as software quality indicators for object-oriented systems [1] [5]. The Hackystat analyses use CK metrics to give its users insights about the software they create.

El-Emam's study [2] tries to find the optimal class size for object-oriented software. Perry's study [6] is observing people to evaluate how time is spent during software development. Both studies are interesting, however they give little to no results about how to do software engineering. Results found are not conclusive.

The next section is describing what our research is based on.

3. Environment

This study has been conducted at the University of Hawaii at Manoa, Department of Computer Science during the Fall semester 2002. The research has been done in collaboration with CSDL. The subjects that we used for our study were students in the classes ICS 414 (an undergraduate class) and ICS 691 (a graduate class) during that time.

During a period of three months, the students were involved in various projects entirely implemented in Java. Hackystat collected sensor log data and metrics about the Java programs they implemented.

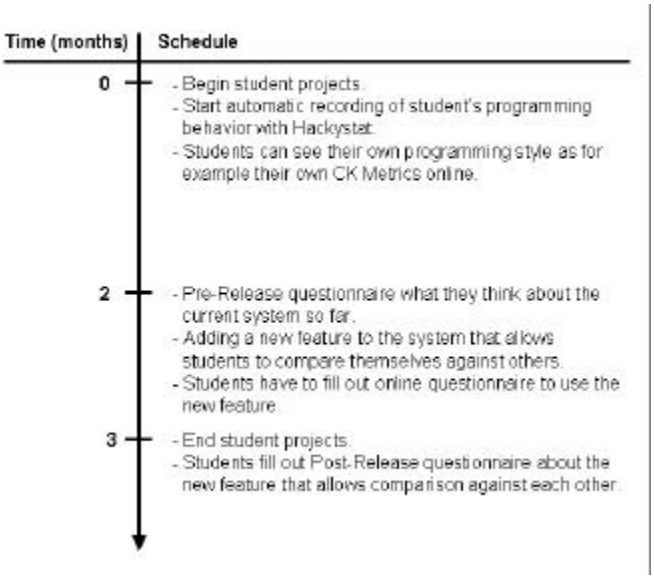


Figure 1: Schedule of Events

The first two months the standard Hackystat analyses were available to each student to analyze their own software development process. In the last month we gave the students the possibility to compare themselves against other students. Figure 1 depicts an insight into the schedule of events taken place.

This research is based on the extension of the Hackystat tool which automatically collects metrics while the students program. Some of the metrics and analysis the Hackystat provides are: active/idle times, CK metrics, effort (hours), JUnit test invocations, number of methods, lines of code, etc. These metrics are collected by sensors placed in technologies such as in JBuilder and Ant. A central Hackystat server is receiving the data and stores data locally as XML files and makes the log files available through a web interface. See figure 2 for details.

In the first two months our subjects were restricted to

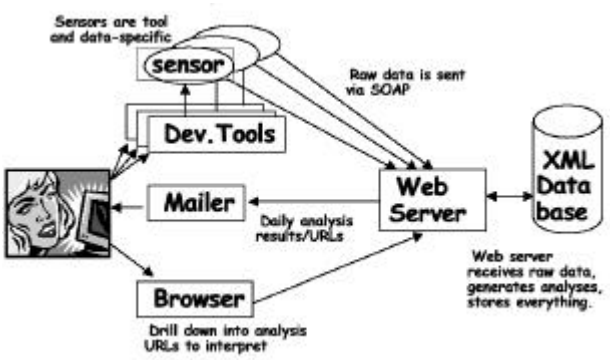


Figure 2: Hackystat Architecture

view their own log data without relation to other students. Analyzing CK metrics data or active/idle time adds insights to the personal software development process, however comparing these metrics to other students allows them to think about their own programming behavior and helps them to better understand how they relate to other students.

The extensions that were made to Hackystat include creating new analyses and an online questionnaire. The comparison analyses that were created are based on the standard Hackystat analyses. For example, Hackystat currently includes an analysis on the amount of active and idle time that the user gathers any given day. We extended this analysis to allow the user to compare his active/idle times against a group of users. Currently we have implemented the following comparison analyses: active/idle, hours worked, code added, code removed and the CK metric suite. For each of these analyses we include three charts:

- *Chart 1*: Comparison of current user to all users in Hackystat.
- *Chart 2*: Comparison of current user to a group of users with the same demographic value as the current user. If the demographic is *education level* and the current user is graduate student her programming characteristics are compared to other graduate students.
- *Chart 3*: Comparison of all demographic groups within Hackystat. If the chosen demographic is *education level*, then graduates and undergraduates get compared to each other.

These charts were chosen to allow the user to make two important comparisons, which are aimed to help the user to gain information about their programming on two different levels.

First, Chart 2 allows the user to compare a specific metric to metrics that are from her peers. Therefore, the comparison provides the user with information of average value of that specific metric for her peers. Using this information the user can then decided if he is deficient or sufficient in that specific metric. An example of this comparison is the hours comparison; the user feels that she is not getting the grades that she desires on programming assignments. She can then use the hours comparison analysis to check if she is not working the average amount of hours than that of her peers.

The second important comparison is made with Chart 3. Chart 3 compares the different demographics within Hackystat. This chart allows a user to compare how different groups compare to each other. An example of this comparison is the CK metric lines of code comparison; the user has just learned Java and feels that his classes have too many lines of code. He can use the CK metric lines of code comparison to compare his lines of code versus graduate students. The user can then aim to reduce his lines of code of the classes he writes to what graduate students do. An important note is that the assumption is that graduate students have more experience writing Java programs than new students of computer science.

These comparisons are made possible by defining groups of users with the same demographic characteristics. However, these groups are not stored in Hackystat explicitly; rather they are attributes that the individual user contains. The extension of an online questionnaire to Hackystat allows each user to voluntarily add these demographic attributes to their user profile. Upon a comparison analysis Hackystat searches for users with the specific demographic the comparison requires. The online questionnaire contains the following questions:

- Months of experience in programming (in Java and other languages, at school, at work and for personal usage)
- Education level (graduate, undergraduate, highest level of ICS class done)
- Programming interest

The next section shows how we conducted our experiment.

4. Experimental design

There are several ways to evaluate the comparison analyses that we have created. They include looking at various comparison analyses to see the type of information that can be obtained from them and a questionnaire to subjectively measure the usefulness of the comparison charts.

The pre-release questionnaire evaluation will give us insights to a users' views of the current metrics and analyses that the standard set of Hackystat analyses provide. A second, post-release questionnaire will be given to assess what they liked or didn't and what they would like to have added. We also gave them the opportunity to comment about our research. This questionnaire evaluation is intended to subjectively evaluate the usefulness of the comparison analyses. The questionnaires will be given to the students in the classes ICS 691 and ICS 414. The total number of students in these two classes is 30. The period which separates the two questionnaires will be two weeks, to give the students enough time to look over the comparison analyses that have been implemente

The next section during our study.

5. Results

Our results are based on a pre and post-release questionnaire that students filled out regarding our comparison analysis, which we added to Hackystat. The students gave their opinion about the Hackystat system before and after we added our feature. Further we analyzed the data collected by Hackystat and compared different demographics against each other. The students had to fill out an online questionnaire.

5.1. Pre and Post-Release Questionnaire

The results of the pre and post-release questionnaires varied greatly among different students. The pre-release questionnaire asked about the usefulness of the original Hackystat analyses.

After a period of two weeks, during which the students had the opportunity to view the comparison analyses, the post-release questionnaire was handed out. The results of our analysis is depicted in Table 1. The meaning of the values are available in Table 2. Also included in the questionnaire were questions asking the students about what type of comparison analyses they would mostly like to have. The results are listed in Table 3.

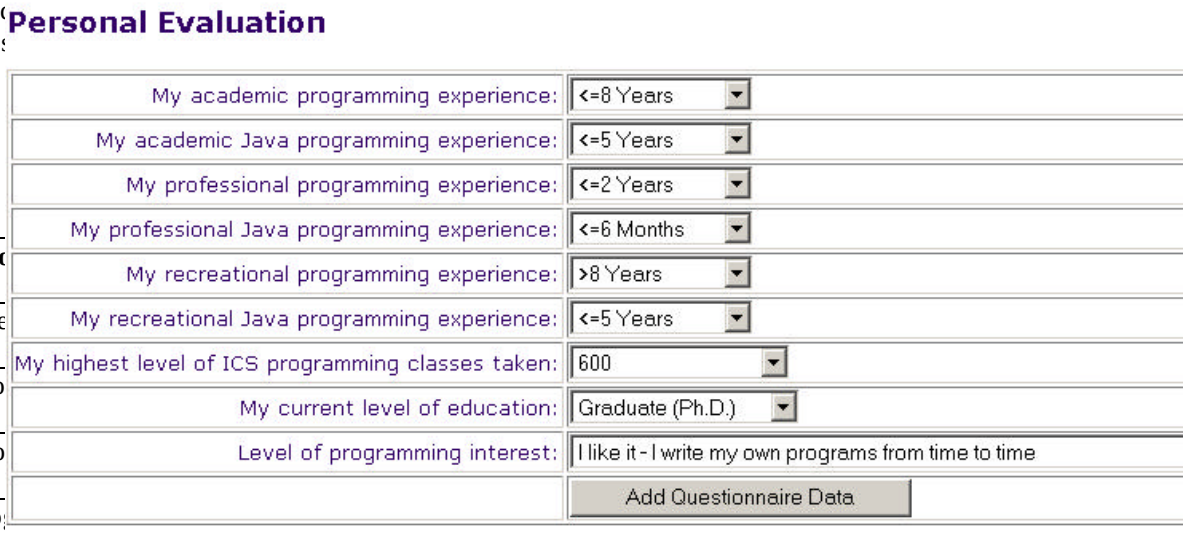


Figure 3: Online Questionnaire

Questi		
The Hackystat analyse useful to me		
I can use Hackystat to program		
I can use Hackystat to programming		
I know where my pro place me in relation to other programmers by using the Hackystat analyses		

Table 1. Average Value of Pre-Release and Post-Release Questionnaire

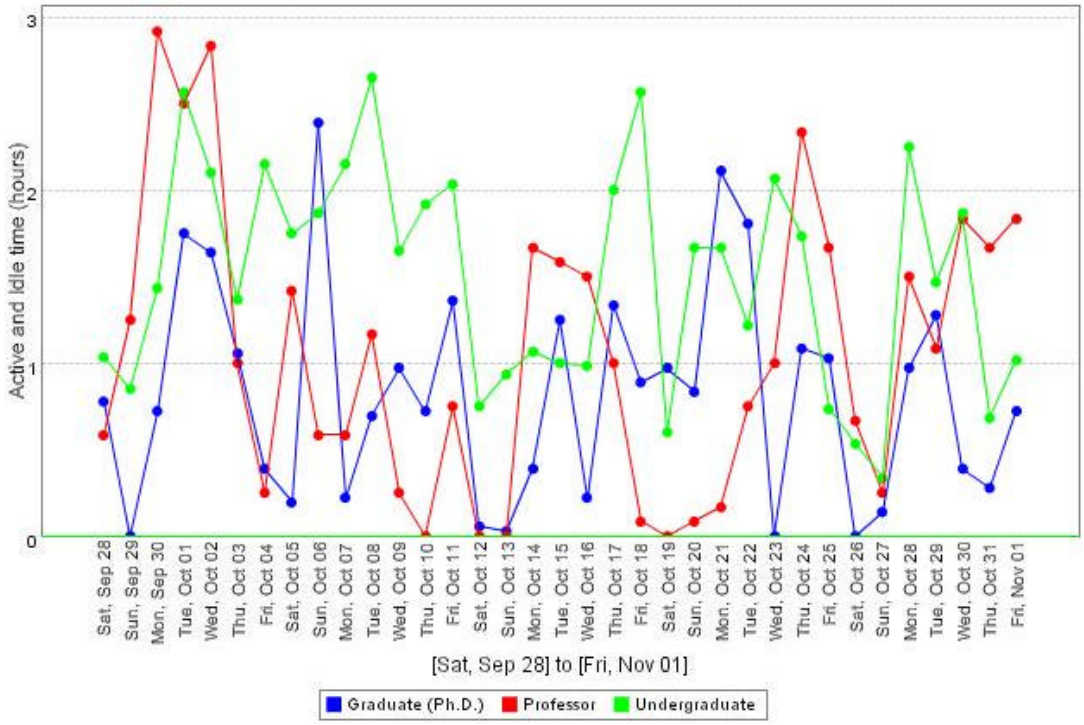
Question	Selected answers (answers are paraphrased)
What analyses would like to be able to compare against other Hackystat users?	LOC (lines of code)
	LOC per hour
	All if possible.
	How fast others program doing a similar project
	Active / Idle time
	CK Metrics
	Complexity of programs, time spent programming, method-level coverage, and unit testing
	Code churn

only 6 of them really used the system and could give us accurate answers.

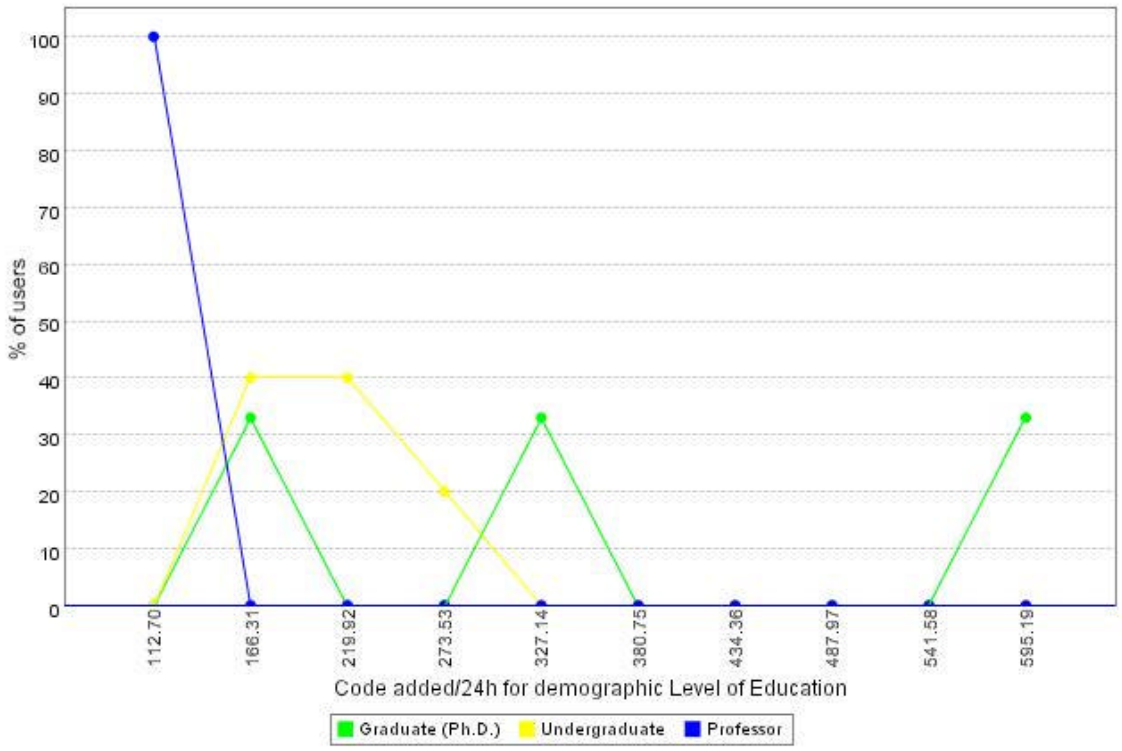
5.2. Demographic Comparison

To do analyses based on different demographics, we used an online questionnaire. See figure 3. Programmers can define their own experience, education level and interest in programming. We were able to get 9 people to fill out the questionnaire. Which of 5 were undergraduate students, 3 were Ph.D. and 1 professor. These out of 30 people that participated in our research. Furthermore we provided 3 charts as described in section 4.

The chart that is interesting for this paper compares different demographics against each other. Questions arose such as:



Graph 1: Active Time Comparison

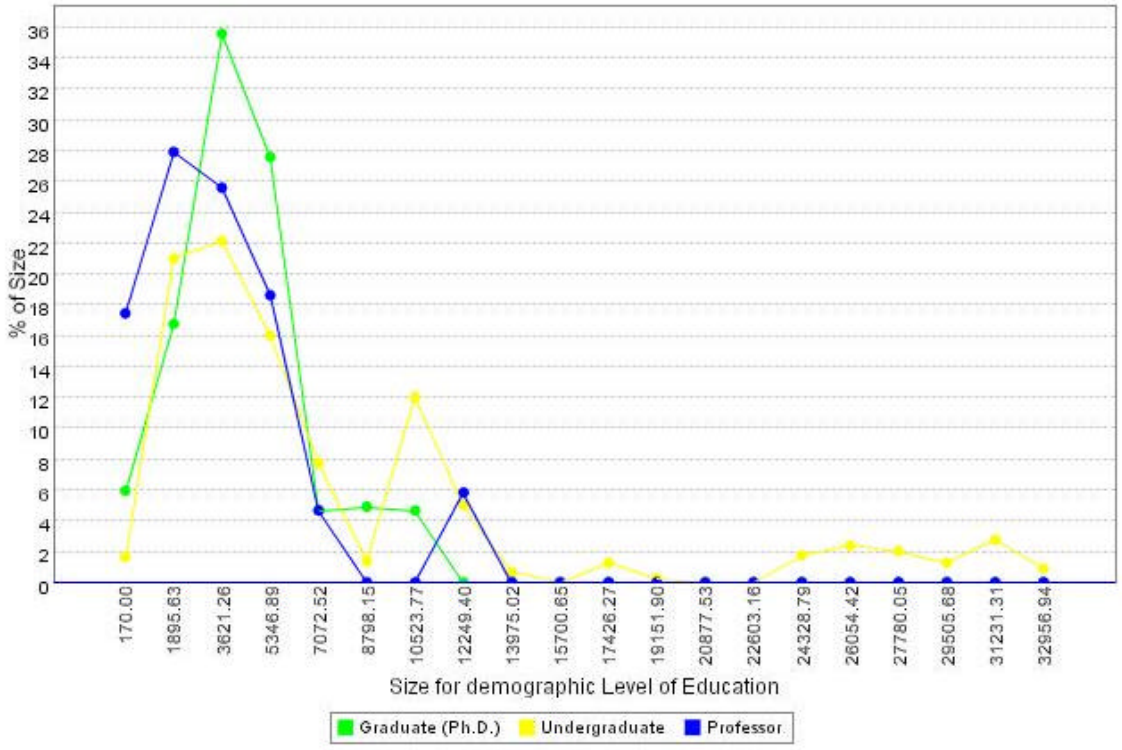


Graph 2: Code Added within 24h

?? Are graduate students better than undergraduates?

?? Can experience be measured in a collegiate setting?

Graph 1 shows a comparison of people during one month of software development. The chosen demographic is level of education. The graph shows the hours spent programming during that time. Changing the



Graph 3: CK Metrics (Size in Bytecode)

demographic and the time period doesn't return any difference. The graph doesn't give any indication about the optimal way of spending time.

Graph 2 shows a comparison with the lines of code added within 24 hours of active programming. The chosen demographic is again level of education. Each group of the demographic, such as undergraduate students, sums up to 100%. With only 9 subjects in the chart, the results are not conclusive. Another analysis with lines of code removed doesn't return any results either.

One of the most interesting parts of our research was to do the CK Metrics comparison. What are the different characteristics of people creating software? Are less experienced software developers programming in a different way? *Graph 3* shows the distribution of Java class sizes produced by different developers. The size is the bytecode size of compiled Java code[1].

As visible in the chart, there is a peak in the graph distribution at around 3500 bytes. The distribution goes from 0 to around 15000 bytes for most programmers. See figure 4 for details. The categories of software developer can be divided into three basic groups:

- ?? *Unexperienced Programmers* – People with no or very little experience in programming.
- ?? *Medium Experienced Programmers* – People with two years or less experience in programming. Undergraduates.
- ?? *Experienced Programmers* – People with a couple of years experience in programming. Higher level of education. E.g. Ph.D. or Master's degree.

One interesting fact on the side. Some of the students developing software created classes of nearly 100000 bytes in size.

Analysis of CK Metrics values other than class size didn't return any conclusive results:

- ?? *WMC (Weighted Methods per Class)* – Peaks of the distribution appear between 3 and 9.
- ?? *CBO (Coupling Between Objects)* – Peaks of the distribution appear between 6 and 18.
- ?? *RFC (Response For Class)* – Peaks of the distribution appear between 10 and 30.
- ?? *DIT (Depth of Inheritance Tree)* – There is not enough data available to make an analysis.
- ?? *NOC (Number Of Children)* – There is not enough data available to make an analysis.

There are no differences visible within a demographic group for the CK metrics mentioned above. Less experienced programmers show the same traits as those with more experience. Undergraduate students show the same traits as graduate students. With only 9 people taking part in our research it was not possible to get conclusive results.

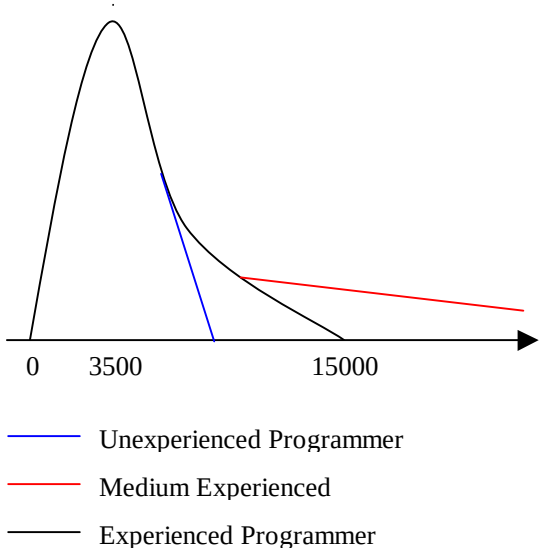


Figure 4: Approximate Size Distribution

The evaluation section gives a critical analysis of our results found.

6. Evaluation

Evaluating the pre and post-release questionnaire, the results found are not conclusive. Only 6 people that filled out our post-release questionnaire really used our system for comparison analysis. So our results are only partly valid for these questionnaires.

Interesting however is table 3. It shows the analysis that is most interesting for students. For our research we didn't implement graphs for method level coverage or unit tests. We implemented most of the other comparisons requested in table 3. However, it seems that from the people that requested the graphs, only 6 out of 30 students really used our system to try the comparison analyses.

One of the problems doing our project in a class environment was of not having an incentive for the students using our system. They didn't get any benefits for testing it. Rather, they spent time for our research and didn't get their own work done. So at the very end, participation for our project was pretty low. Students wouldn't even spend 5 minutes of their time to logon to our system and fill out our questionnaire in figure 3.

At the very end we were able to convince 9 people to logon to our system and fill out the online questionnaire.

[1] ? bytes of compiled source code correspond to 1 line of compiled source code.

The results described in section 5.2 all relate to these 9 subjects of which 5 were undergraduate students, 3 were Ph.D. and 1 professor.

The charts shown in this paper all relate to the demographic "level of education". It correlates with the demographics "experience level in programming". "Highest level of classes taken" or "Interest in Programming" didn't show any correlation at all. Even though people seemed to be interested in programming, they didn't show similar programming traits as found in other demographics such as education level. There was no clear distinguishing between the different subgroups of these demographic possible.

The subjects participating in our research were doing different projects. Some created completely new software products, others just extended existing programs. Some people extended the Hackystat system as we did. Other groups created sensors for tools such as CVS, Eclipse or Forte for Java. Others created analysis tools.

The next section is providing ideas how to proceed with further research.

7. Contributions and future directions

Our pilot study showed some results that could support product and process improvement in software engineering. The research has to be seen as a first step. As mentioned in the evaluation section, some problems arose during our study.

One important point in doing further research is to have more people participating. It is difficult to make clear statements with only a small group of subjects. Furthermore, subjects should be evenly distributed with level of education or experience level. This makes analysis easier.

The comparison analyses to Hackystat that we created didn't supply any comparison analyses for code level coverage or unit tests. Unit tests can be used to verify the fault proneness of a software product. Both metrics would allow to improve the quality of a program.

One possible next research step could be to get a couple of people together and let them do the same kind of programming task. As in our research we didn't have the opportunity to let all our subjects create the same system. The results that we got may not be completely reliable. Everybody that did programming worked on a different project. Creating a completely new program or changing an existing one might return different results. It could be interesting to see the difference between working on new software or extending one that already exists.

Many students didn't test our system because they didn't have enough time. They didn't compare themselves against other students. Some people mentioned they were just to lazy to use Hackystat all the time. One way to solve that would be to send out a weekly e-mail to the student to let them know where they stand against other people. So they wouldn't need to logon to a web site to see their personal metrics.

Hackystat could be used as a learning tool. So far the system is collecting metrics and allows to view them but doesn't make any suggestions how to program better. Using comparison analyses, Hackystat could be used to find and verify good programming traits. This would be the first step. The second step would be to let people know through e-mail where they stand against other programmers and how they can improve. An unexperienced programmer can be teached through e-mail how to develop software better. Undergraduate students can be compared to graduate students and given advice what to change in their process. E.g. e-mails stating "your class sizes are too big" or "your DIT value is too high for your system produced" could help improve products and processes in software engineering.

Using Hackystat, everything can be automated, so overhead for students or professors is small. Required is a registration in the Hackystat system and installation of the sensors in the programming tools such as JBuilder, Emacs or Eclipse.

8. Acknowledgements

We thank Philip Johnson for his advice and support in writing this paper. Furthermore we thank all the students from ICS 413 and ICS 691 in Fall 2002 who participated in our research.

9. References

[1] K. El-Emam, "Object-Oriented Metrics: A Review of Theory and Practice", *NRC/ERB-1085*, NRC/CNRC, Canada, March 2001.

[2] K. El-Emam, S. Benlarbi, N. Goel, W. Melo, H. Lounis, S. N. Rai, "The Optimal Class Size for Object-Oriented Software: A Replicated Study", *NRC/ERB-1074*, NRC/CNRC, Canada, March 2000.

[3] P. M. Johnson, A. M. Disney, "A Critical Analysis of PSP Data Quality: Results from a Case Study", *CSDL Lab*, University of Hawaii at Manoa, August 1999.

[4] W. S. Humphrey. "Pathways to Process Maturity: The personal software process and team software process." *SEI Interactive*, June 1999.

[5] V. R. Baslii, L C. Briand, W. L. Melo. "A Validation of Object-Oriented Design Metrics as Quality Indicators" *IEEE Transactions on Software Engineering*, VOL. 22, No. 10, October 1996.

[6] D. E. Perry, N. A. Staudenmayer, L. G. Votta, "Understanding and Improving Time Usage in Software Development" *At&T BellLaboratories, USA, Massachusetts Institute of Technology Sloan School of Management, USA*, 1995.

[7] Collaborative Software Development Laboratory (CSDL), University of Hawaii at Manoa, http://csdl.ics.hawaii.edu

[8] CSDL, *Hackystat tool*, http://hackystat.ics.hawaii.edu