The Human Factor in Computer Science and How to Teach Students to Care: An Experience Report

Janet Siegmund and Sven Apel

University of Passau, Germany

Abstract: The human factor plays a crucial role in software engineering, so software engineers should pay sufficient attention to it. In this paper, we present our experience with teaching software-engineering students to care about the human factor. In particular, we report on a course that we conducted at the University of Magdeburg, in which we applied explorative and interactive techniques to teach the basics of human-behavior measurement. In summary, we received mostly positive feedback of the students, and found that after the course, students are able to properly take care of the human factor.

1 Importance of the Human Factor

In the late 1960s, software developers had to face increasingly complex software, eventually leading to the software crisis. In part, the crisis was caused by the fact that software was not developed for humans, but for computers. As Dijkstra phrased it in his 1972 Turing lecture "The humble programmer":[1]

> [O]ne programmer places a one-line program on the desk of another and either he proudly tells what it does and adds the question "Can you code this in less symbols?" [...] or he just asks "Guess what it does!"

In these days, programming was seen as art—understandability or maintainability of source code was not the primary concern. Furthermore, usability of programs was not an issue, because only few, highly trained people worked with computers. Today, almost everyone uses computers regularly, for example, when using a smart phone. Even globally dispersed team members can collaborate on a single project with the help of collaborative software systems. Thus, the role of humans, either alone or as a group, either as a developer or as a user, is important.

Unfortunately, human behavior is difficult to predict; we cannot easily predict whether two humans in the same situation behave the same—we cannot even predict whether one human behaves the same if encountering the same situation twice. Instead, we need to conduct empirical studies, in which we observe people when they work with source code

[1]`http://dl.acm.org/citation.cfm?id=355604.361591`

or when they use a program, so that we can predict how new programming languages or tools or features of a program affect the human who is using it.

However, conducting empirical studies in software engineering is rather uncommon. Only recently, empirical investigations, especially those that focus on the human factor, become more and more common. Before that, researchers who have developed new techniques with the goal of improving comprehensibility of source code or the usability of user interfaces, often argued only with plausibility arguments about why the technique or interface is more comprehensible or more usable. In practice, the claimed benefits may not hold or are difficult to evaluate.

For example, in Word 2000, Microsoft introduced adaptive menus.[2] Instead of a fixed order, menu items arrange according to their frequency of usage, so their order changes during usage. This way, the designers hoped to increase the efficiency of using Word, because frequently used menu items were always on top. However, in practice, this did not work, because with adaptive menus, the location of menu items appears to be non-deterministic. Thus, users look for a menu item in the wrong place, and are slower with adaptive menus.

One reason for the reluctance of conducting empirical investigations is that they require considerable effort and knowledge, which is often not taught during computer-science education. Thus, researchers often underestimate the effort and importance of a sound study design, or depend on trained experimenters.

To address this problem, we need to train software-engineering researchers in empirical methods. Although the call to integrate empirical methods in the computer-science curriculum is not new [Bra05, fC01], empirical methods are only in few universities part of the software-engineering curriculum, for example, at the Karlsruhe Institute of Technology (Walter Tichy), Freie Universität Berlin (Lutz Prechelt), or University of Marburg (Christian Kästner).

We designed and held a course at the University of Magdeburg, which was initially based on course developed by Christian Kästner at the University of Marburg, in cooperation with Stefan Hanenberg. For our teaching philosophy, we changed several aspects of the course, including the order and emphasis of the material for our course design. In this paper, we report our experience with teaching empirical methods to computer scientists, for which we adopted explorative and interactive teaching methods. Our contribution in this paper is twofold:

- We share our experience with teaching experience a course on empirical methods, including comments of enrolled students.
- We discuss the contents and teaching methods of our course to help other researchers and teachers in designing a course similar to ours.

Our overarching goal is to raise the awareness of the human factor in software-engineering education as well as software engineering in general. Furthermore, we want to initiate the

[2]http://humanized.com/weblog/2007/03/05/are_adaptive_interfaces_the_answer

path toward a common course description and teaching material, so that software engineers can properly address the human factor in software engineering.

2 Content of the Course

In this section, we present the content of the teaching course, which we held at the University of Magdeburg in the winter term 2012/13, and which we will hold at the University of Passau in the summer term 2014. In the course, we have covered software measurement, the importance of the human factor, systematic planning and conducting of experiments, as well as quantitative and qualitative methods.

2.1 Software Measurement

In this part, we have covered typical computer-science topics, that is, topics, that the typical student has encountered during his/her education at the University of Magdeburg. We have started with measuring software based on software measures, because that is close to students' previous experience. We have introduced different software measures, such as lines of code and cyclomatic complexity, and let students compare different software systems based on software measures. During the measurement process of software, typically no unsystematic measurement error occurs, that is, the same software, measured with the same tool, always has the same software measures. In this context, the measurement process is straightforward and intuitive.

As next step, we introduced performance measurement, that is, students should measure the performance of software systems. This introduces the concept of measurement error, as measuring the same software system under the same condition does not necessarily lead to the same results. Thus, to measure one software system, several measurement runs should be conducted (which is what all students intuitively did). Furthermore, we did not specify performance further, leaving the operationalization to the students.

2.2 The Human Factor

Based on these first steps, students have experienced that even without the human factor, having sound measurement procedures already requires considerable effort. When the human factor is added, there is even more variation. For example, different developers have different levels of programming experience, are familiar with different programming languages or domains, and might prefer one tool over the other. Thus, different individuals introduce measurement bias, which has to be taken care of.

Hence, with this first part, we have shown students the necessity for sound methodological training in empirical research. Even for empirical investigations without the human factor, such as performance measurements, significant bias can be introduced. In our course, we continued with introducing a structured way to conduct empirical research, which we present in the next section.

2.3 Conducting Empirical Investigations Systematically

To minimize bias in empirical investigations as much as possible, we need a structured way to conduct them. To this end, we can divide empirical investigation into five stages: objective definition, design, conduct, analysis, and interpretation. Next, we give a short overview of each stage.

Objective Definition. First, during the objective definition, the goal of the empirical study is defined. This includes stating research hypotheses or questions and *operationalizing* the constructs of interest. The research hypotheses drive the further design and prevent "fishing for results", that is, playing around with the data until something interesting is found.

Experimental Design. Second, we need to develop the experimental design, which defines how we evaluate the hypotheses or how we answer the questions. A major obstacle in this stage is to control for *confounding parameters*, which may severely bias the results. In our experience, handling confounding parameters is the most difficult and most often neglected part of empirical studies, because researchers simply are not aware of them.

Conduct. Third, the experiment is conducted. In this phase, despite all careful planning, numerous things can go wrong. For example, experimenters can influence participants, participants deviate from their instructions, or there might be power failures. All deviations should be thoroughly documented.

Analysis. Fourth, the data obtained from the experiment conduct needs to be analyzed. In our experience, researchers often do not know what to do with data beyond computing average scores or visualizing data. However, average scores are not sufficient to answer the question whether a difference is real or whether data correlate. To this end, we need to conduct statistical tests. Furthermore, we teach standard visualization techniques, such as box plots and histograms, which is important to present the results.

Interpretation. Last, we need to interpret the data, which goes beyond accepting or rejecting hypotheses or answering research questions. Instead, we need to state what the results mean. Is a new programming paradigm better for programming experts, but not for novices? Should we really start teaching programming with this programming paradigm? Thus, the results need to be put into perspective beyond the experiment.

In our course, we taught students to follow this procedure when designing experiments. Next, we introduced quantitative and qualitative methods to measure the human factor.

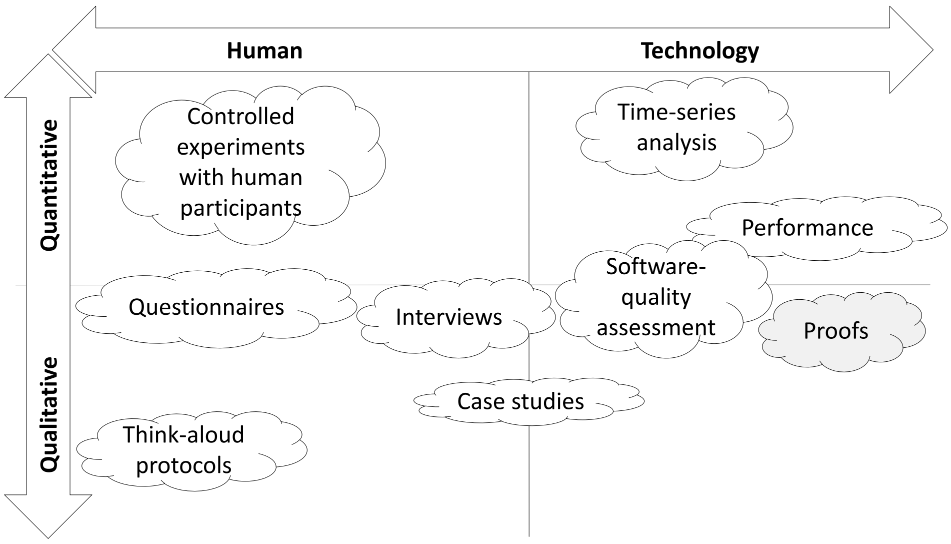

Figure 1: Mindmap on qualitative and quantitative methods in conjunction with the human/technology factor for the empirical-methods course. Elements with a gray background were omitted in the course, but shown for completeness.

2.4 Quantitative and Qualitative Methods

To measure the human factor, there are different quantitative and qualitative methods. For a better overview, we developed a mindmap for students, and showed them at the beginning of each lecture, where the topic belongs to. In Figure 1, we show the map.

We started with quantitative methods, because in discussions with fellow researchers and students, we learned they are more intuitive for computer-science students than qualitative methods. Specifically, we discussed how to design controlled experiments, in which several participants are observed, and in which only few data points per participants are measured. For example, we explained how to measure program comprehension based on response time and correctness for a group of 20 or more participants. In this part, we also introduced typical analysis procedures and their logic, from descriptive statistics (e.g., mean, standard deviation, box plots, confidence intervals), to significance testing (e.g., χ^2 test, Student's t test, ANOVA).

After quantitative methods, we introduced qualitative methods, in which few participants are observed in detail. In particular, we introduced interviews, case studies, and think-aloud protocols. Furthermore, we discussed analysis techniques for qualitative data, from grounded theory to card-sorting techniques, including reliability measures, such as Cohen's Kappa for interrater reliability.

With teaching these techniques, we enabled students to select an appropriate empirical method for an evaluation of the human factor, so that students are able to conduct an empirical study in their Bachelor's or Master's thesis.

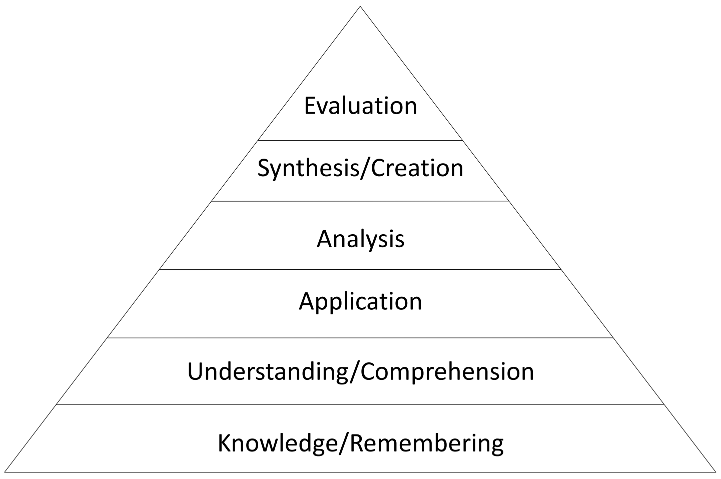

Figure 2: Levels of learning according to Bloom [BEF+56].

For the course design, we relied on our experience when discussing empirical studies with colleagues and students. In particular, we often found that people need some time to grasp the difficulties of soundly measuring the human factor, so we decided on the order that we felt is most intuitive for computer-science students. Furthermore, we selected methods that we often encountered in the literature, so that students learned a representative set of empirical techniques.

While in this section, we concentrated only on the content we taught, we explain in the next section the teaching methods, that is, *how* we ensured that students understand and apply suitable empirical methods.

3 Teaching Methods

Teachers face the problem of getting students to deeply understand and care about the contents of a course. In Figure 2, we show the six levels of learning, as described by Bloom and others [BEF+56]. At the lowest level (*knowledge/remembering*), students take in data, remember it, and recite it. At the second level (*understanding/comprehension*), students can explain information in their own words. At the third level (*application*), students are able to apply information in a new way. Students at the fourth level of learning (*analysis*) can break information down into its parts, and at the fifth level (*synthesis/creation*), can combine the parts to build a new structure. At the highest level (*evaluation*), students can judge the value of information. In our course, we aim at the highest level of learning, *evaluation*, so that students can select an appropriate method for any empirical question.

So, how do we achieve the highest level of learning? In our course, we used a combination of explorative and interactive teaching methods, which we explain in the next sections.[3]

3.1 Exploration

During the complete course, we let students explore empirical methods for themselves, with the goal that they notice the problems when using an intuitive approach. For example, regarding performance measurement, we divided the students in groups of three to four students each and let them evaluate the run-time performance of different sorting algorithms in different programming languages. We did not explain to students how performance is reliably measured, or how to control for confounding parameters, but we let them follow their intuition. Afterwards, the student groups compared their results, and found that no group would trust the results of another group. We concluded this exercise with a guided discussion that led to the conclusion that we need sound empirical methods to control for confounding parameters, so that we get trustworthy and practically applicable results.

Another explorative method was that students read and discussed papers that report controlled experiments. Based on these examples, students learned how to address typical problems of empirical research (e.g., how to operationalize variables, how to control for confounding parameters), as well as how typical pitfalls can be avoided.

To ensure the highest level of learning, students conducted their own empirical study. To this end, students could select any question they were interested in and design an experimental plan to evaluate the question. After feedback on the experimental plan (to ensure that students would not run into too much trouble), students conducted the experiment, analyzed the data, and wrote a report (available on the course's website: `http://wwwiti.cs.uni-magdeburg.de/iti_db/lehre/emcs/2012/`). Thus, students experienced themselves the complete process of one empirical study, which we believe showed them the difficulty of sound empirical studies, and also sensitized them for the importance of the human factor.

3.2 Interaction

Another feature of our course is interactive teaching, that is, involving students in the lecture, not just presenting them information. In addition to exploration, a lot of interactive methods exist. That includes asking students during the lecture to suggest solutions to presented problems, were appropriate, after a few-minutes discussion with their neighbor (referred to as *buzz groups*).

To highlight how to involve students in the teaching process, we discuss how we introduced the systematic procedure to conduct empirical investigations (cf. 2.3). For brevity, we fo-

[3]There are several books (e.g., [Sil96]) and online sources (e.g., University of Zurich (`http://www.hochschuldidaktik.uzh.ch/hochschuldidaktikaz.html`)) that give comprehensive overviews of explorative and interactive teaching methods.

cus only on the first stage. First, we introduced a running example, that is, we used the research question *Do comments make source code more comprehensible?* as starting point. Then, we started with the objective definition, in which we introduced the terms independent and dependent variable, as well as hypothesis. After explaining what an independent variable is, we asked students to name the independent variable in the example (which is *comments*). We explained what operationalization is, and then asked students to operationalize comments (e.g., comments can either be present or not present, or comments can be incorrect or correct). We did the same for the dependent variable (i.e., *comprehension*), which can be operationalized with letting participants fix a bug, and then measure the correctness or response time of the bug fix. The faster the response time, the better comprehension should have taken place. After defining the variables, we talked about hypotheses, and that they must be falsifiable, that is, if they are wrong, we must be able to show that. We asked students to decide whether the research question is suitable as hypothesis (which it is not, because it is too unspecific), and then, after a short discussion with their neighbor, give examples of more suitable hypotheses (e.g., *Incorrect source-code comments slow down program comprehension*). For the remaining stages, we used the same pattern, that is, explaining information to students for 5 to 10 minutes, and then asking them to apply the information right away. This way, they deepened their understanding and can better memorize information, which is a well-studied phenomenon [CT75].

We also included other interactive methods, of which we present a couple of examples, with which we had a particularly good experience in our course. First, we used *black stories/situation puzzles* as interactive methods. Situation puzzles are a game, in which a host explains a situation to the players, and the players have to find out by asking only yes-or-no questions how this situation emerged. In the course, we presented students the conclusion of an experiment, for example, that expert and novice programmers show equivalent program comprehension, and students should come up with the experiment plan. In this case, there were different source-code snippets, and several of them contradicted the expectation of expert programmers, making them as incorrect and slow as novice programmers [SE84]. This way, students learned to look at experiments from a different angle, which we believe helped them to get a deeper understanding.

As second example, we used the interactive methods *world café*, *vernissage*, and *student award*. In the world café, students designed an empirical study in a group of about four students for a given research question (e.g., *How do students learn programming?*), and prepared a flip chart to present their results to the other groups, which was then presented (vernissage). This way, students applied the taught techniques and deepened their understanding in discussions. After the vernissage, students selected the best experimental design (student award), which helped them to critically review and understand the experimental plan of the others.

To summarize, with explorative and interactive teaching methods, we ensured a higher level of learning, compared to the classical way of merely presenting the information to students. And this is also what our students said in the evaluation, which is conducted for every lecture (see next section).

3.2.1 Evaluation

To ensure high-quality teaching, it is custom at the University of Magdeburg to conduct an evaluation at the end of the semester. In this evaluation, students can give comments about what they like and dislike about a course, and how it can be improved. We received ten answers for our course, which can be summarized as follows:

1. First, we found that most of the students liked the interactive teaching style, because it motivated them to analyze the presented information actively (instead of only listening to information).
2. Second, most students also liked that they could explore information based on real examples, and that they could try out the empirical methods and analysis techniques on short examples as well as on their own project.
3. Third, in contrast to the positive feedback, there are also few students who did not like the explorative part, especially when they had to conduct their own study.

Despite the negative feedback, we feel confident that our students profit from this way of teaching. Thus, when we conduct the course again, we will continue using the explorative and interactive teaching techniques.

Considering the projects of the students, we observed a surprisingly high quality of studies. For example, one project explored factors influencing personal web-search behavior. To this end, the students assessed several factors that might influence web-search behavior, such as experience with using the web, frequency of using computers or the web, or education, and designed web-search tasks of different difficulty levels. In the analysis, students correlated the performance in the web-search tasks with the assessed factors, and created hypotheses based on their analysis. During all experimental stages, students used the proper methods and draw the correct conclusions from their results. The complete report is available on the project's website (`http://wwwiti.cs.uni-magdeburg.de/iti_db/lehre/emcs/2012/projekte/FabianAnton.pdf`).

Considering the positive evaluation results and the high quality of the empirical studies, we believe that our course prepares computer-science students well to soundly measure the human factor in computer science.

3.2.2 Course Description

Most university courses require a course description as part of the examination regulations for a study course. In Table 1, we show how we described the course as example for other teachers.

We suggest to offer the course to graduate students, because the course requires reading and understanding scientific papers, which is often too advanced for undergraduate students. Furthermore, we can plan the course for 5 or 6 ETCS, which translate into 60 hours of attending the lecture and participating in experiments, and 90 (5 ETCS) or 120 (6 ETCS) hours of homework assignment, including the project. Since in the project, students apply

Issue	Content
Course name	Empirical methods for computer scientists
Level	Master
Teaching	4 hours per week
Effort	5/6 ETCS
Recommended prerequisites	Knowledge on software engineering
Learning goals	After this course, students: • Know empirical methods for evaluating research questions • Can assess the validity of scientific statements • Can apply a suitable empirical method for evaluating research questions in a bachelor's or master's thesis
Content	Results in computer science often aim at better quality, lower costs, better maintainability, or better comprehensibility. To be able to evaluate these claims, we need to use empirical methods, which are the content of this course. For illustration, we use examples from software engineering and programming languages. Contents include: • Scientific method, proofs, empirism • Rigorous measurement of performance, including benchmarks, case studies • Controlled experiments with developers • Statistical background knowledge
Relevant for examination	• Participation in lectures • Participation in experiments of other students of this course • Evaluating a self-selected research question • Completing homework assignments • Oral examination

Table 1: Excerpt of the syllabus for the empirical-methods course.

the taught techniques on their own study, the time students invest for the project should not be reduced, but only the time for the homework assignment. We recommend that students are familiar with software engineering, because most of the examples are in this domain. This course description is intended to help other researchers integrate a similar course at their university.

4 Summary

In this paper, we highlighted the importance of the human factor in software engineering. We argued that one reason for the underrepresentation lies in the negligence of empirical methods in the computer-science curriculum. We shared our experience from our course we held at the University of Magdeburg, so that other researchers are encouraged to introduce a similar course at their university. This way, we hope to have raised the awareness for the necessity of teaching empirical methods to computer scientists, and to have helped other researchers to introduce a similar course at their university. The material of the course is available online (`http://wwwiti.cs.uni-magdeburg.de/iti_db/lehre/emcs/2012/`).

Acknowledgement

Thanks to Christian Kästner and Stefan Hanenberg for fruitful discussion on the design of the course.

References

[BEF$^+$56] B. Bloom, M. Engelhart, E. Furst, W. Hill, and D. Krathwohl. *Taxonomy of Educational Objectives: The Classification of Educational Goals. Handbook 1: Cognitive Domain.* David McKay, 1956.

[Bra05] Grant Braught. Teaching Empirical Skills and Concepts in Computer Science Using Random Walks. In *Proc. Technical Symposium on Computer Science Education (SIGCSE)*, pages 41–45. ACM Press, 2005.

[CT75] Fergus Craik and Endel Tulving. Depth of Processing and the Retention of Words in Episodic Memory. *Journal of Experimental Psychology*, 104(3):268–294, 1975.

[fC01] Joint IEEE Computer Society/ACM Task Force for CC2001. Computing Curricula 2001. `http://www.acm.org/education/curric_vols/cc2001.pdf`, 2001.

[SE84] Elliot Soloway and Kate Ehrlich. Empirical Studies of Programming Knowledge. *IEEE Trans. Softw. Eng.*, 10(5):595–609, 1984.

[Sil96] Mel Silberman. *Active Learning: 101 Strategies to Teach Any Subject.* Pearson, 1996.