The Human Factor in Computer Science

Janet Siegmund

janet.siegmund@uni-passau.de

Abstract: The human factor plays a crucial role in software engineering. In our work, we highlight the importance of the human factor during the software life cycle. Furthermore, we discuss how it can be soundly assessed and evaluated, such that common pitfalls can be avoided.

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 group, either as developer or as user, is very important.

Unfortunately, human behavior is non-deterministic; we cannot easily predict whether two humans in the same situation behave the same—we cannot even predict whether one human behaves the same. Instead, we need to observe people when they work with source code or when they use a program, which we do in empirical studies.

However, conducting empirical studies in computer science is not common. Especially the role of developers is neglected. Instead, researchers who have developed new techniques with the goal of improving comprehensibility of source code or the usability of user interfaces argue with plausibility arguments why the technique or interface should be more comprehensible or more usable. In practice, the claimed benefits may not hold or are difficult to verify.

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

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

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

during usage. This way, designers hoped to increase efficiency of using Word, because often used menu items were always on top. However, in practice, that did not work out, because with adaptive menus, the location of menu items appears non-deterministic. Had users been observed while using adaptive menus in empirical studies, the disadvantage of changing location of menu items could have been revealed.

One reason for the reluctance of conducting empirical investigations is that it requires 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.

In the presentation, we give an introduction into empirical study design and discuss common pitfalls and how they can be avoided. To this end, we discuss the five stages of conducting empirical investigations: Objective definition, design, conduct, analysis, and interpretation. In the next paragraphs, we give a short overview of each stage and present common pitfalls.

First, during the objective definition, the research hypotheses or questions have to be defined and the constructs of interest have to be operationalized. The research hypotheses drive the further design and prevent "fishing for results", that is, playing around with the data until we find something interesting. The constructs of interest also have to be specified, so, for example, what exactly does program comprehension or usability mean and how can we measure it?

Second, we need to develop the experimental design, which defines how we evaluate the hypotheses or how we answer the questions. A major problem in this stage is to control for *confounding parameters*, which may severely bias the results. So, do we recruit programming experts or novices? Do we recruit users who we know prefer a certain interface? How can we motivate participants to behave normally? In our experience, handling confounding parameters is the most often neglected part of empirical studies, because researchers simply are not aware of them.

Third, despite all careful planning, a lot can go wrong when executing the experiment. For example, experimenters can influence participants, or participants deviate from their instructions.

Fourth, data 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, this does not answer the question whether a difference is real or whether data correlate.

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 presentation, we show ways to address the raised issues based on standard techniques that are well established in psychological research for more than hundred years. We, the computer scientists, can profit from this experience. After the presentation, the listeners are able to appreciate the delicateness of empirical research.