Human Factors in Programming and Software Development

MARY BETH ROSSON

Virginia Polytechnic Institute and State University, Blacksburg ⟨rosson@vt.edu⟩

Behavioral studies of programming, which emerged in the late 1970s, were among the earliest in the field of human-computer interaction (HCI). HCI is the study of humans interacting with computing systems, and well before the appearance of modern interactive computer applications, programmers were using text-based command and programming languages to solve complex problems on computers. Cognitive scientists are intrigued by the complex and open-ended nature of software design problems; practitioners are eager to analyze and improve the productivity and reliability of software development. This article discusses issues and findings of behavioral research in this area.

NORMATIVE STUDIES OF PROGRAMMING

As a user population, programmers present many challenges for scientific study [Curtis 1988]. They vary wildly in ability and style, so much so that some have argued that the most effective way to "teach" design skills is to identify individuals with inherent talent and simply nurture them. It is exceedingly difficult to recruit professional programmers for empirical studies, and historically most such work has studied computer science undergraduates. As programmers, university students are at once unrepresentative of the target population, yet still marked by great variation in individual ability.

The tasks of software development also vary a great deal—the factors influencing twenty minutes of code comprehension may have no influence on a program-generation task taking several hours. Pragmatic problems such as these have frustrated attempts to produce normative accounts of the factors influencing programming. For example, while the *prima facie* intuition is that structured programming techniques must certainly aid programming, relevant empirical findings are mixed, due to the variability introduced by individual differences, languages, tasks, and so on [Curtis 1985].

ANALYSES OF PROGRAMMER COGNITION

Many studies have analyzed the mental activities of individual programmers designing, building, testing, or comprehending code [Hoc 1993]. These studies often employ qualitative methods, studying a few carefully selected individuals in great detail by recording and analyzing verbal "think-aloud" protocols. Programmers' behaviors and comments are organized and tabulated and used as evidence for the cognitive representations and strategies in play. Studies of this sort often exploit the "unrepresentativeness" of university students by contrasting their emerging knowledge structures and strategies with those of more advanced programmers.

Important theoretical constructs in the analyses of programmer cognition are the plan—a hypothesized mental structure corresponding to a stereotypical program component, such as a conditional expression or a counter mechanism—and the strategy an expert follows in using his or her plan knowledge. As expertise develops, programmers exhibit more "plan-like" behavior:

they decompose design problems in a more orderly and balanced fashion (especially if they possess domain experience). Their code composition is organized by these stereotypical structures; sometimes a focal piece of a plan is generated first and the details filled in later, while at other times a more linearized "read-out" of the complete plan occurs. When experts are asked to comprehend or test programs, they use their general and domain-specific knowledge to develop hypotheses about program content, and then search for "beacons" (cues) that signify the presence of corresponding code structures.

PROGRAMMING-IN-THE-LARGE

A complementary vein of work considers programming-in-the-large—realistic software development projects, typically carried out by teams working within a software development organization. A major focus is on team dynamics that improve productivity: for example, a structured team with a chief programmer is suitable for large, simple projects on a tight schedule, but complex projects requiring high creativity are better served by teams with high individual flexibility. Team programming can be viewed as distributed cognition, where the product is viewed as the result of a complex system of programmers, their internal mental activities, and their shared externalized task representations [Hutchins (to appear)].

The organizational context of software development also has important effects on programming-in-the-large. Professional programmers spend considerable time communicating with others in their organization, both individually and as part of a group. Thus the analysis of communication problems—for example, groups not realizing they are even supposed to communicate, misunderstandings about a shared issue, conflicting views from different groups, or changes in project personnel [Curtis 1988]—is a key element in understanding how to better support the software development process.

PROGRAMMING LANGUAGES AND TOOLS

When confronted with novel problems, expert programmers often adopt an opportunistic working style, decomposing the problem in a flexible fashion that enables them to develop partial solutions along the way and to recognize possibilities for code reuse. Green [1993] discusses three dimensions of programming languages and tools that influence a programmer's working style: *viscosity*: the degree of resistance to local changes; *premature commitment*: the extent to which a decision must be made before its consequences can be seen; and *role expressiveness*: the ease of discovering the purpose of software components.

A well-encapsulated object-oriented language such as Smalltalk improves viscosity, aiding refinement and the composition of partial solutions. Structured editors can simplify code generation, but may force premature decisions about control and data structures. Languages that allow names for extended variables increase role expressiveness, thus facilitating comprehension and the reuse of existing code.

FUTURE DIRECTIONS

A topic currently attracting great interest is the potential impact of the object-oriented (OO) paradigm on programming and software development [Hoc 1993]. Proponents claim that the OO technique, which emphasizes problem-oriented software abstraction, should increase the naturalness of software development and support a more seamless integration of analysis, design, and implementation activities. As yet little empirical work has assessed this; most reports document the difficulties of teaching OO design and programming to experts trained in functional decomposition and procedural programming. However, significant changes are occurring in computer science education, with an increasing number of universities and colleges

integrating OO techniques into their curricula, setting the stage for more meaningful assessments in the future. At the same time, researchers are applying OO concepts to the problems of end-user programming, for example, exploring the metaphor of intelligent objects as active agents assisting users with simple programming tasks.

REFERENCES

CURTIS, B. 1985. *Tutorial: Human Factors in Software Development*. IEEE Computer Society, New York.

CURTIS, B. 1988. Five paradigms in the psychology of programming. In *Handbook of Human-Computer Interaction*, M. Helander, Ed. North Holland, Amsterdam, 87–106.

HUTCHINS, E. L. *Distributed Cognition*. MIT Press, Cambridge, MA. (to appear)

GREEN, T. R. G. 1993. Programming languages as information structures. In *Psychology of Programming*, J.-M. Hoc, T. R. G. Green, R. Samurcay and D. J. Gilmore, Eds. Academic Press, New York, 117–138.

HOC, J.-M., GREEN, T. R. G., SAMURCAY, R., AND GILMORE, D. J. 1993. *Psychology of Programming*. Academic Press, New York.

ROSSON, M. B. AND ALPERT, S. R. 1990. The cognitive consequences of object-oriented design. *Human-Comput. Interaction 5*, 345–379.