Language Design: A Cognitive Science Approach (Full Presentation)

Michael E. Hansen, Andrew Lumsdaine, and Robert L. Goldstone

Indiana University

Factors in Language Design. Programming language design and evolution are often driven by largely technical factors, such as changes in hardware (e.g., multi-core, GPU) and support for particular paradigms (e.g., object-oriented, functional). The impact of design decisions on usability, however, is rarely evaluated in a scientific manner [7]. Instead, widely-used languages are either advanced by a committee of experts (C++, Java), or by a single expert (Python, Ruby). Feedback from the community is important in both cases, but the decision to accept or reject a given feature ultimately lies with one or more experts. This process works well except in cases where experts disagree – i.e., *technical matters alone are not enough to judge a feature*. For these situations, there is currently no objective method for resolving disagreements.

A Quantitative Cognitive Model. We believe a promising avenue of research is to develop a quantitative cognitive model of the programming process. The ultimate goal of this research is to provide a scientifically-grounded means of ranking code by its **cognitive complexity** [3]. In other words, the desired model should be able to predict the difficulty of understanding code for a programmer with specific characteristics (e.g., expertise with the language or problem domain) and for a particular task (e.g., reading, modifying, debugging). The parameters of the model should be cognitively meaningful, relating to known limitations and architectural features of the brain [10] as well as the programmer's existing knowledge base [19].

With such a model in hand, the impact of language changes could be evaluated based on the distribution of simulated code rankings over a portion of the model's parameter space. While the model must be informed by user studies in order to mimic human data, it should provide information beyond simple correlations between language changes and user performance. A good cognitive model must help explain **why**, and not just what. Judgements based on model simulations might take the form of *"feature X negatively impacts novice users because it requires them to simultaneously attend to too much information"* or *"feature Y positively impacts experts during code reading because their visual search strategies are highly tuned to an aspect of Y."*

Existing Metrics and Models. It is common to associate the "complexity" of a program with particular textual features of its source code (e.g., cyclomatic complexity [16], program effort [15]) and the relationships between the program's functions or objects [9]. These traditional complexity metrics are useful in regression models whose primary goal is to predict the maintainability or fault proneness of code, but they do not provide direct insight into the cognitive aspects of programming. More importantly, traditional complexity metrics do not provide a model capable of explaining the impact of language design decisions *on people*. Rather, these metrics quantify the impact on the code itself (e.g., more or fewer tokens), leaving the psychological impact on the programmer to be inferred.

For the purposes of explanation alone, a plethora of qualitative cognitive models of the programming process exist in the field of Program Comprehension. The Cognitive Dimensions of Notation framework, for example, emphasizes the trade-offs between different aspects of programming languages (e.g., hidden dependencies, redundancy, premature commitment) [14]. The Stores Model of Code Cognition focuses on the visual, spatial and linguistic abilities of programmers [8]. The Task Memory Model [18] imports recent research on working memory from Cognitive Neuroscience, stressing the importance of the brain's many distinct memory sub-systems. While these models provide a foundation for additional research and experimentation, they do not themselves offer objective, quantitative predictions. In order to ground program comprehension models, we believe it is necessary to develop a comprehensive computer model of the programming process, spanning the gap between perception and concepts in the problem domain. Recent research into the memory of domain experts has been fruitful in uncovering important architectural aspects of human memory and learning in general.

Expert Memory and Chunking. A quantitative cognitive model of the programming process does not exist yet. However, decades of research into the memory systems of experts in different domains (e.g., chess, physics, electronic circuits) has laid much of the groundwork [19, 11]. Experiments have revealed that experts in many domains outperform novices due to their vast mental libraries of *chunks*: knowledge units of the domain that are stored in long-term memory [4, 5, 13]. Having these units available lets experts quickly and efficiently store representations of the problem at hand, allowing them to overcome the limitations of working memory [17]. Although novices and experts share the same memory constraints (5 ± 2 chunks), experts are able to hold more domain information in working memory because their chunks are larger. Beyond encoding the immediate problem, experts are also believed to possess *templates*: long-term memory retrieval structures with "slots" that facilitate rapid learning of new domain information [12].

Early experiments with Pascal programmers provided strong evidence for the existence of perceptual chunks [19]. More recent experiments have confirmed these findings, and have demonstrated robust expertise effects, such as the ability of experts to recall scrambled code better than novices.

Computer simulations of a code recall experiment[1] using the Chunk Hierarchy and REtrieval STructures (CHREST) cognitive architecture revealed that the expertise effect could be elicited purely by increasing the number of code samples the model was allowed to observe beforehand [11].

Conceptual chunking has also been observed in programmers at the program level, where internalized code schemas strongly drive expectations [7]. In a series of clever experiments, it was shown that breaking implicit coding "rules of discourse" caused expert programmers to consistently misinterpret the intended use of a particular variable. When answering a fill-in-the-blank experiment, where a small portion of a program's code is missing, experts often filled in the missing pieces with prototypical answers (i.e., initialize the variable to zero), despite evidence that this was incorrect [19]. Program schemas have also been shown to induce distortion effects during recall [6], where experts "remember" prototypical code instead of actual code (i.e., `j` is recalled as `i` in a `for` loop).

What's Missing. The experiments and simulations above are a place to start, but a great deal of work remains to be done. The majority of research with chunking theory has been done in the domain of chess, where problem solving is largely a perceptual processes. Programming, while having a strong perceptual component, requires a blend of perceptual and conceptual processes. Semantic errors in a program, for example, are not always discoverable by simply observing the source code (properly formatting code can make some more apparent).

Research has been done on how programmers relate pieces of code to entities in the problem domain (called the Concept Assignment Problem [1]), but it has not been viewed from the perspective of chunking theory. Similarly, others have investigated the conceptual roles of program variables [2], but with a focus on education rather than the cognitive processes involved in role detection and assignment. We believe a quantitative cognitive model that spans perceptual and conceptual levels will provide a solid foundation for language usability design decisions.

References

[1] T. Biggerstaff, B. Mitbander, and D. Webster. Program understanding and the concept assignment problem. *Communications of the ACM*, 37:72–82, May 1994.

[2] C. Bishop and C.G. Johnson. Roles of variables and program analysis. In *Proceedings of the 5th Finnish/Baltic Conference on Computer Science Education*, 2005.

[3] S. Cant, D. Jeffery, and B. Henderson-Sellers. A conceptual model of cognitive complexity of elements of the programming process. *Information and Software Technology*, 37:351 – 362, 1995.

[4] W. Chase and H. Simon. Perception in chess.

[5] A. de Groot, F. Gobet, and R. Jongman. *Perception and memory in chess: Studies in the heuristics of the professional eye*. Van Gorcum & Co, Assen, Netherlands, 1996.

[6] F. Détienne. Program understanding and knowledge organization: the influence of acquired schemata. In *Cognitive ergonomics*, pages 245–256. Academic Press Professional, Inc., 1990.

[7] F. Détienne. *Software design—cognitive aspects*. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[8] C. Douce. The stores model of code cognition. In *Psychology of Programming Interest Group*, 2008.

[9] K. El-Emam. Object-Oriented metrics: A review of theory and practice. *NRC Publications Archive*, 2001.

[10] E. Feigenbaum and H. Simon. Epam-like models of recognition and learning. *Cognitive Science*, 8(4):305 – 336, 1984.

[11] F. Gobet and I. Oliver. A simulation of memory for computer programs. *Department of Psychology, ESRC Centre for Research in Development, Instruction and Training, University of Nottingham (UK), Technical report*, 74, 2002.

[12] F. Gobet and H. Simon. Templates in chess memory: A mechanism for recalling several boards. *Cognitive Psychology*, 31, 1996.

[13] F. Gobet and A. Waters. Computational modelling of mental imagery in chess: A sensitivity analysis. In *Proceedings of the 30th Annual Meeting of the Cognitive Science Society*, Mahwah, NJ: Erlbaum, 2008.

[14] T. Green. Cognitive dimensions of notations. In *People and computers V: proceedings of the fifth conference of the British Computer Society Human-Computer Interaction Specialist Group, University of Nottingham, 5-8 September 1989*, page 443. Cambridge University Press, 1989.

[15] M. Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Ltd, 1977.

[16] T. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2:308–320, 1976.

[17] G. Miller. The magical number seven, plus or minus two: some limits on our capacity for processing information. *Psychological Review*, pages 81–97, Mar 1956.

[18] C. Parnin. A cognitive neuroscience perspective on memory for programming tasks. In *In In the Proceedings of the 22nd Annual Meeting of the Psychology of Programming Interest Group (PPIG)*, 2010.

[19] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Trans. Software Eng.*, 10, 1984.

[1] In a typical code recall experiment, programmers are briefly presented with a small program and asked to recall as much as possible after it is hidden. In general, the percentage of code correctly recalled is a function of expertise, with prototypical pieces being recalled first.