Programming Languages and Learning

Contributors:

Andreas Stefik
University of Nevada, Las Vegas

Patrick Daleiden
University of Nevada, Las Vegas

Diana Franklin
University of Chicago

Stefan Hanenberg
University of Duisburg-Essen

Antti-Juhani Kaijanaho
University of Jyväskylä

Walter Tichy
Karlsruhe Institute of Technology

The goal of this pamphlet

This pamphlet is designed to provide an overview of recent evidence on human factors evidence in programming language design. In some cases, our intent is to dispel myths. In others, it is to provide the result of research lines.

The community of scholars advocating for and participating in evidence-based programming language design is growing. This sheet is not comprehensive, but covers several of the broad trends and highlights a number of empirical studies that have been performed.

Computer science for all

In the United States, one broad trend is toward computer science education for all. As such, students in K-12 or beyond are learning to program. Despite this trend, the programming language community is highly fractured, with thousands of products used for a variety of purposes.

Teachers and students participating in CS for all may not have the training to evaluate the feature sets of various language products or their impact on people (e.g., students, professionals, those with disabilities). Modern computer science should significantly re-examine issues involved in the programming language wars.

Programming languages make up the foundation of software technology. As we push toward computer science for all, the design of our languages should be based increasingly on evidence.

The Language Wars are Insufficiently Studied

A systematic search and selection process of peer-reviewed papers found only 156 publications (reporting 137 empirical primary studies and 19 secondary studies of empirical primary studies) somehow comparing language design decisions, broadly construed, with respect to human-factors efficacy measures. When limited to studies comparing features, and excluding certain research designs that arguably aren't empirical, the number goes down to 65 primary studies. When further limited to controlled experiments, 35 primary studies remain; further restrict to randomized trials, and we have only 22 primary studies from the 1950s up until 2012 [2].

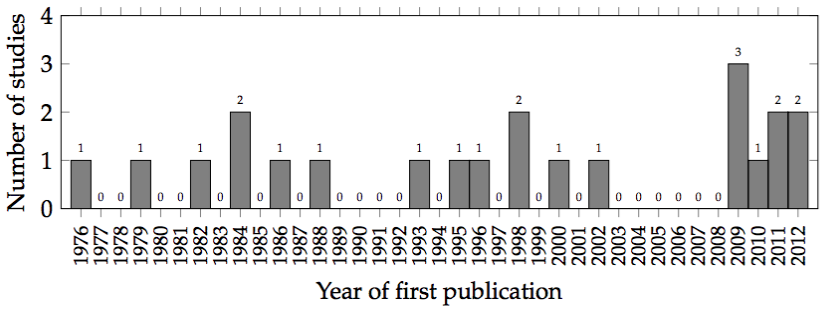


FIGURE 10 The number of randomized controlled experiments in the core per publication year

References:

[1] David Weintrop and Uri Wilensky. 2015. Using Commutative Assessments to Compare Conceptual Understanding in Blocks-based and Text-based Programs. In Proceedings of the eleventh annual International Conference on International Computing Education Research (ICER '15). ACM, New York, NY, USA, 101-110.

[2] Kaijanaho, A.-J., Evidence-Based Programming Language Design: A Philosophical and Methodological Exploration. PhD Diss., Information Technology Faculty, University of Jyväskylä, 2015.

[3] Lars Fischer and Stefan Hanenberg. 2015. An empirical investigation of the effects of type systems and code completion on API usability using TypeScript and JavaScript in MS visual studio. In Proceedings of the 11th Symposium on Dynamic Languages (DLS 2015). ACM, New York, NY, USA, 154-167.

[4] Stefan Endrikat, Stefan Hanenberg, Romain Robbes, and Andreas Stefik. 2014. How do API documentation and static typing affect API usability? In Proceedings of the 36th International Conference on Software Engineering (ICSE 2014). ACM, New York, NY, USA, 632-642.

[5] Victor Pankratius and Ali-Reza Adl-Tabatabai. 2014. Software Engineering with Transactional Memory Versus Locks in Practice. Theor. Comp. Sys. 55, 3 (October 2014), 555-590.

[6] Andreas Stefik and Susanna Siebert. 2013. An Empirical Investigation into Programming Language Syntax. ACM Transactions on Computing Education 13, 4, Article 19 (November 2013), 40 pages.

[7] Lutz Prechelt, Barbara Unger, Michael Philippsen, and Walter Tichy. 2003. A controlled experiment on inheritance depth as a cost factor for code maintenance. J. Syst. Softw. 65, 2 (February 2003), 115-126.

[8] Christopher D. Hundhausen, Sean F. Farley, and Jonathan L. Brown. 2009. Can direct manipulation lower the barriers to computer programming and promote transfer of training?: An experimental study. ACM Trans. Comput.-Hum. Interact. 16, 3, Article 13 (September 2009), 40 pages.

[9] Franklin, D., Hill, C., Dwyer, H., Hansen, A., Iveland, A., and Harlow, D. Initialization in Scratch: Seeking Knowledge Transfer, SIGCSE, 2016.

Static vs. Dynamic Typing

Experiments have shown a positive impact of statically typed languages

A number of experiments have been performed that show a positive impact of static type systems over dynamic type systems. This benefit was found in situations where developers had to use an API that was new to them. In two studies, the positive type system effect was compared with the documentation effect (also, the effect achieved by code completion). In both experiments, the positive impact of type systems was significant and much larger than the documentation or code completion effect. Static typing also showed a positive impact on debugging time for type errors comparing Java and Groovy (where Groovy was used as a dynamically typed Java). The results showed that the Java group was significantly faster fixing the type error [3,4].

Software Transactional Memory (STM)

STM can reduce programming time and prevent synchronization bugs

In a study by Pankratius and Adl-Tabatabai, student teams developed a desktop search engine with either locks or STM. The STM teams were among the first to have an operational implementation and spent less than half the time debugging segmentation faults, but had more problems tuning performance and implementing queries. Code inspections were suggestive that STM code was easier to understand than locks code, because the locks teams used many locks (up to thousands) to improve performance [5].

Notation

The notation used in programming languages has a large impact on novices

In a study of six programming languages using novices, one randomized controlled trial found that accuracy rates for certain C-style languages (Perl, Java) were not significantly higher than a language with randomly generated keywords and symbols, while languages that deviated from this style did (Quorum, Ruby, Python). Statistical procedures called Token Accuracy Mapping now exist that can predict which tokens contribute, positively or negatively, to the overall effect [6].

Inheritance

Inheritance depth is not a significant driver of software maintenance effort

Code maintenance effort is not strongly correlated with inheritance depth. Instead, factors such as the number of methods to change and the experience of the maintainer appear more related. In one controlled experiment, Prechelt et al. studied the impacts on maintenance with five levels of inheritance depth compared to “flattened” versions of the same program with three and zero levels. The effort level was most highly correlated with the number of methods that needed to be understood for the change, followed by the maintainer’s experience and then inheritance depth, which made little difference. Reducing or eliminating this depth does not seem to have much effect on the effort required for understanding software [7].

Early Learning and Visualization/Blocks

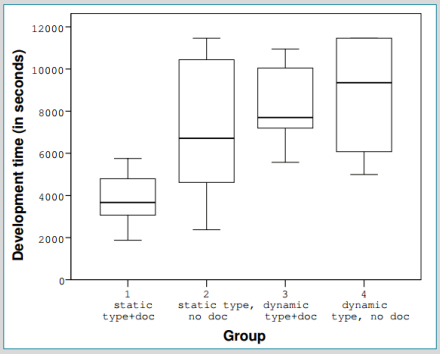
Curriculum for elementary education and early learning

When analyzing a curriculum, look at what earlier skills lead to success in the large and teach those in isolation before introducing more complex programming tasks. While most first grade students cannot program "real" programs, students can still learn to think in ways that lead to strong computational thinking skills. They already do activities that have some relationship to computing - color-by-number pictures, following algorithms, classifying objects by type. The key is to identify how to frame those activities to make connections with computational thinking, as well as identify what bridge activities could come between existing activities and more mature activities [9].

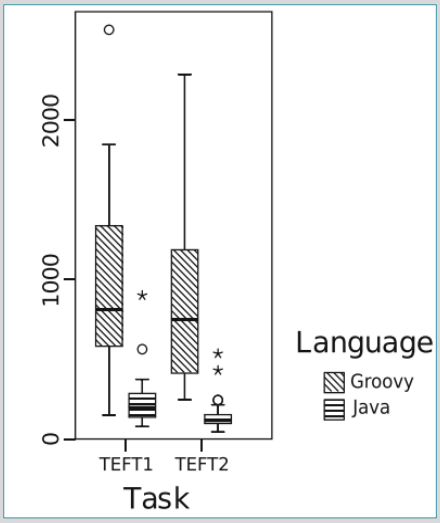
Block Languages may impact learners differently than text

Visual Block-based languages, though lacking some of the functionality of text-based languages, have been shown to lead to knowledge transfer to text-based languages under some conditions. Initialization of state, however, is one concept that is challenging for students within block-based languages and does not match text-based languages. [1, 8, 9].

Static vs. Dynamic Typing



Debugging Time of Type Problems



Token Accuracy Mapping of Syntax