Relationship of Early Programming Language to Novice Generated Design

Tzu-Yi Chen
Math and Computer Science
Pomona College
Claremont, CA 91711 USA
+1 909 607 8651
tzuyi@cs.pomona.edu

Alvaro Monge
Computer Eng. and Computer Science
California State University, Long Beach
Long Beach, CA 90840 USA
+1 562 985 4671
amonge@csulb.edu

Beth Simon
Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0404 USA
+1 858 534 5419
esimon@cs.ucsd.edu

ABSTRACT
What measurable effect do the language and paradigm used in early programming classes have on novice programmers' ability to do design? This work investigates the question by using data collected from 136 "first competency" students as part of a multi-national, multi-institutional study of students' approach to and attitudes toward design. Analysis of a number of surface characteristics of their designs found strikingly few differences between designs produced by students at schools that teach using objects-early, imperative-early, and functional-early paradigms. A similar lack of difference was found between students at C++-first and Java-first schools. While statistically significant differences are found for three characteristic comparisons across language and paradigm, these results seem to have little meaning for teaching given the complexity of the null hypotheses tested in those three cases. In particular, for the following design characteristics no statistically significant differences across language or paradigm of early instruction were found: attempt to address requirements, type of design produced, number of parts in design, recognition of ambiguity in design, and connectedness of design.

Categories and Subject Descriptors
K.3.2 [Computers and Education]: Computer and Information Science Education; D.2.10 [Software Engineering]: Design

General Terms
Design, Languages.

Keywords
Multi-institutional, novice, first competency, CS1, design, paradigm, programming language.

1. INTRODUCTION
Many computer science educators have an opinion, sometimes strong, on the language or paradigm (eg, imperative-first, objects-first, functional-first) that is best for early instruction (eg, [9,17]). Studies reporting on the basis for these opinions often consider the effect of first language or first programming paradigm on the difficulty of teaching, and presumably learning, various concepts (eg, [8,11,14,16]). Similarly, panels at SIGCSE have debated the merit of different approaches (eg, [1]). However, whether the ease of teaching different concepts has a greater impact on early programming ability or on student ability to design is unclear.

The distinction is worth noting, particularly in the context of research that analyzes problem solving performance as being composed of the ability to design and the lower-level skill of programming. Earlier work has found that beginning students have difficulty solving problems at a level expected by instructors [12], and that they seem to have difficulty evaluating the quality of a design [13]. Given that some believe a first programming course should be more focused on teaching principles of successful design than on teaching programming in any one language [8], this paper asks whether the language and/or paradigm taught in an early programming course affects the designs created by beginning students. In particular, the impact of instructional language or paradigm on the ability to program or on the ease of teaching concepts is ignored; the focus is on design and the higher-level thinking it requires.

The results here come from data collected in a multi-national, multi-institutional study of student-generated software designs. The focus is on designs produced by first competency[1](FC) students who were exposed to different programming languages and/or programming paradigms early in their coursework. Their designs were evaluated on a range of surface characteristics such as the type of representation produced, the number of parts in that representation, whether they indicated interactions between the parts, how many of the stated design requirements they attempted to meet, and whether they recognized ambiguity in the design process. While these characteristics touch on the quality of the designs produced, no attempt was made to compare the designs using a standardized metric of quality.

As discussed in depth in later sections, detailed statistical analyses reveal strikingly little difference between FC students who were exposed to different languages or to different instructional paradigms. This might suggest either that CS educators need not worry about the effect of early language and paradigm choices on design ability, or that more work is needed to understand the expected effect of language and paradigm on design.

SIGCSE'06, March 1-5, 2006, Houston, Texas, USA.

[1] FC students are defined by the criteria given in [6]: students at an educational level where instructors would expect them to be able to write a basic calculator program.

2. METHODOLOGY

The results reported on in this paper are based on data collected as part of a multi-national, multi-institutional study of student-generated designs [6,15]. In that study over 300 participants at 21 institutions in 4 countries were given a written description of a "Super Alarm Clock" and asked to complete a design of that clock. Each participant committed their design to paper, then described their solution verbally to a researcher, and finally engaged in a prioritization task that elicited what design criteria they considered most and least important in different scenarios.

The data collected consisted of the written artifacts (referred to as design representations), a transcript of their verbal description, and their rankings of different design criteria. In addition, second order data was produced by the researchers involved in the project [6,15]. Some of this data was gathered in a distributed way – that is each researcher coded the data gathered from their institution. This includes categorization of certain surface features of the design representations including the students' attempts to address the requirements of the specification; number of parts identified in the representation or transcript; students' use of hierarchical, nested, or grouping structure in the representation; and students' indication of interaction between parts in the representation. Additionally, researchers determined from the transcripts whether students recognized ambiguity in the design specification. While this type of second order data is appealing in that each design is analyzed by someone who is already somewhat familiar with it, the reliability and reproducibility of the coding between researchers may vary significantly. For example, what one researcher considers two parts another might see one part with a nesting structure. Other second order data that was collected as part of [6,15] does not suffer from the reproducibility problem. For example, to evaluate the type of each representation, a subset of the researchers jointly classified each design as one of the following: graphical, textual, code-based, based on a standard such as UML, or something else ("miscellaneous"). In addition, in some cases researchers were unable to classify designs to their satisfaction. Because of this, some of the following results are based on an analysis of a subset of the 136 first competency students. In all cases, at least 130 students are represented and specifics are discussed in each section.

The analysis in this paper considers the language and paradigm introduced in the first programming class at each first competency student's institution, and looks for correlations between those variables and any of the above characteristics of student designs. Given the diversity of student backgrounds, simply being a student at a school that introduces programming using a particular language and paradigm does not necessarily mean this is how that student was first exposed to programming. However, it does mean that the student was exposed to this language/paradigm approach relatively early.

By considering only the artifacts generated by first competency students, the questions addressed in this paper are fundamentally different from earlier papers based on the same corpus of data. Other papers focused primarily on differences in design between participants at different education levels (e.g. [2,5,6,15]).

3. ANALYSIS

To support the claim that the observed results can be partially accounted for by differences in early language or paradigm, other data on students' background was also analyzed. For example, the number of languages that students reported having some previous familiarity with was analyzed. Both Java and C++ students claimed familiarity with similar numbers of different programming languages (mean=3.4 for C++ and 3.47 for Java) and at a similar competency level (about 3 out of 5). Although Scheme (mean=4.14) and SML (mean=6.4) students claimed to know a greater number of languages, one-way ANOVAs at $\alpha = .01$ show that the differences by paradigm and by language are not significant.

Note also that the data is primarily nominal in nature, consisting of categories and counts (e.g., the number of first competency students whose design artifacts were code-based). Furthermore, as a way of handling some second order data, occasionally interval data such as the number of requirements addressed is grouped into subsets (e.g., the number of first competency students who attempted 5-8 of the requirements) and considered as nominal data. Because of this, the primary statistical tool used is the χ^2 test. Unfortunately, this test is less reliable when the counts in any one category are very small, as is the case with data collected for the SML and Scheme languages and for the associated functional paradigm. Because of this, the analysis that follows sometimes details general trends when the numbers are too small for meaningful statistical testing.

3.1 Addressing Requirements

One measure of the correctness of a design is the number of problem requirements that the design attempts to address. Each design was evaluated as attempting to fulfill all requirements (all 9), a partial set of the requirements (5-8), hardly any requirements (1-4), or no requirements (0). 130 students are reported on here -- six students were unable to be classified regarding their attempt to address requirements. As the numbers in Figure 1 might suggest, χ^2 tests reveal no significant difference between C++ and Java languages, nor between imperative-early and object-early paradigm learners when evaluating students' attempts to address requirements. Although the numbers for the functional-early students are too small for statistical analysis, note that all functional paradigm students addressed at least half of the nine requirements.

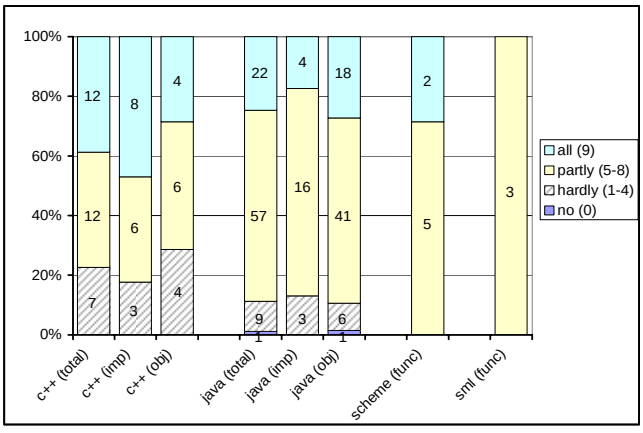

Figure 1: Requirements attempted as a function of early language and paradigm. Numbers in the bars are the counts.

3.2 Different Design Representations

In earlier work analyzing the same corpus of data, statistically significant differences are found in the "marks on paper" representations of designs produced by subjects, with a shift from textual to standard graphical representations with increasing education [15]. The categories of design representation developed include: standard graphical (S), ad-hoc graphical (G), code or pseudo-code (C), textual (T), and mixed (M). Within the FC data no statistically significant difference is found in the types of representations produced as a function of either early programming language or paradigm. Figure 2 (reporting on all 136 students) shows how many FC students produced designs of each type as a function first of paradigm (imperative, object-oriented, or functional), and then of the language used within each paradigm.

Despite the lack of statistically significant differences, the data reveals some trends. First, functional paradigm students did not produce code-based representations, perhaps suggesting that these students are less likely to turn to code to represent their intentions. Additionally, C++ students are about twice as likely to produce code representations as Java students (regardless of paradigm) and about half as likely to produce standard graphical representations (also regardless of paradigm). When considering individual paradigms, OO-early Java students are about twice as likely as OO-early C++ students to produce ad-hoc graphical designs, though imperative-early Java students are less likely to turn to ad-hoc graphical representations than imperative-early C++ students.

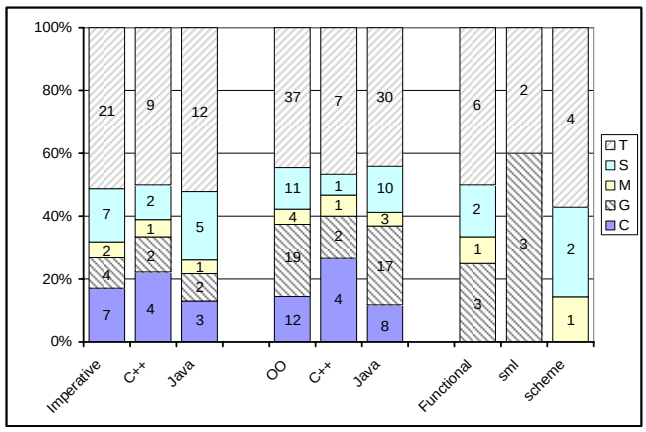

Figure 2: Design representation as a function of early paradigm, and then by language. Numbers of students in each category are shown inside the bars.

3.3 Recognized Expert Characteristics

The design literature identifies a number of design characteristics that are indicative of more expert results. These include dividing the solution into parts ("chunking") [10], presenting the solution in a hierarchical nature [7], showing connectivity or interconnection between parts [3], and recognizing ambiguity in a design problem [4]. This section analyzes the effect of early language and early paradigm on these "recognized expert" design characteristics.

3.3.1 Number of parts

Running a one-way ANOVA on the number of parts evidenced in each representation first with early language as the independent variable and then with early paradigm as the independent variable revealed no differences that were statistically significant. However, C++-early learners generally produced designs with more parts (mean=5.69) than Java-early learners (mean=4.99). Similarly, imperative-early learners generally included more parts (mean=5.45) than object-early learners (mean=5.04), who included more parts than functional-early learners (mean=4.75).

3.3.2 Hierarchical, nested, or grouping structure

Whether or not each design displayed a hierarchal, nested, or grouping structure among the identified parts was determined by the individual researchers who gathered the raw data. As noted previously, this type of second order data is subject to greater bias in interpretation than other kinds of data. In addition, five subject representations could not be categorized, hence the analysis in this section was done on a total of 131 representations.

Using a χ^2 test (α=.01), C++ students were found to be more likely to produce hierarchical designs than Java students, perhaps indicating that the learning of the C++ language itself (compared to Java) encourages students to identify hierarchical, nested, or grouping behavior. Neither OO nor imperative learners were found to be more likely to produce hierarchical designs.

Furthermore, no statistically significant difference was found between OO and imperative learners when looking only at either C++ learners or Java learners. For C++ learners, the null hypothesis that paradigm has no impact on hierarchical design could not be rejected for α<0.18. Neither could the null hypothesis be rejected for Java learners; this is not surprising given Figure 3, which shows the percentages and counts of students evidencing hierarchical, nested, or grouping structure in their design as a function of language and then paradigm. The three bars on the left show that imperative C++ learners are about twice as likely as OO C++ learners to produce hierarchical designs, but that a consistent 15-20% of Java learners produce hierarchical designs regardless of paradigm.

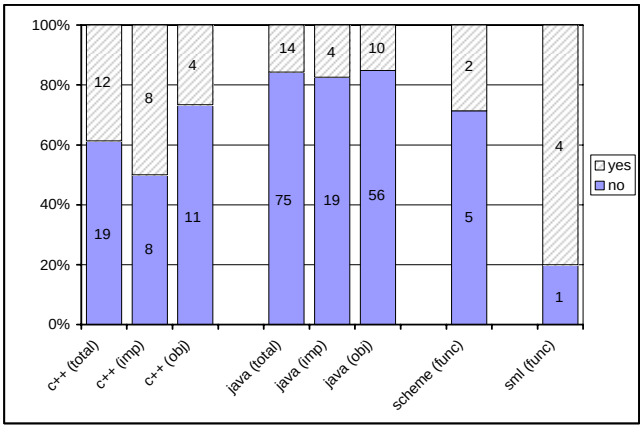

Figure 3: Producing hierarchical, nested, or grouping design as a function of early language and paradigm.

3.3.3 Interaction between components

Considering identified interaction between components in the designs, no statistically significant differences were found across either early programming language or paradigm. 55% of C++ learners and 69% of Java learners had interaction in their designs and 67% of functional learners, 68% of imperative learners, and 64% of OO learners identified interaction. This data is reported

for 133 students – for three students no data on their recognition of interaction was available.

One significant difference at the α=.05 level, not evident in Figure 4, is that C++ learners who identify interaction attempt to address more requirements than C++ learners who do not. The numbers show that only 1 of the 17 C++ students who identified interaction met fewer than half of the requirements, whereas the same is true for 5 of the 12 C++ learners who did not identify interaction. Figure 4 shows that imperative first students who identify interaction satisfy more requirements than imperative-first students who do not (significant at the α=.05 level). Again, the numbers show that 2 of 28 addressed less than half of the requirements, as compared to 4 out of the 11 who did not identify interaction. No similarly significant results were found for Java learners or for OO learners.

Nevertheless, concerning paradigm, Figure 4 shows that students who identify interaction are consistently more likely to attempt at least half of the requirements than comparable students who do not, and that this holds across paradigms. This reinforces the fact that identifying interaction is a sign of design maturity while simultaneously highlighting that first language or paradigm does not seem to affect whether a student indicates interaction in their design.

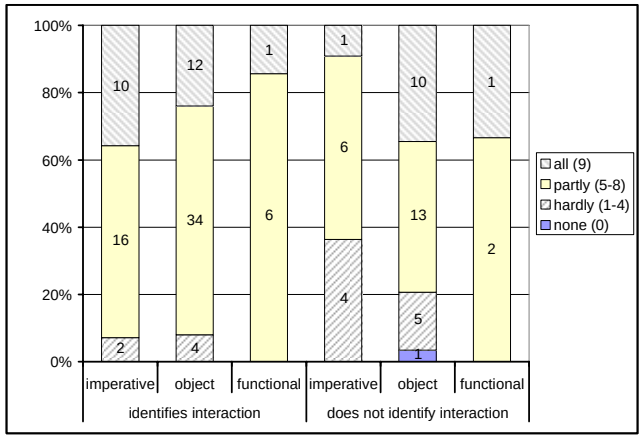

Figure 4: Degree to which students attempted to address requirements as a function of recognition of interaction and then paradigm.

3.3.4 Recognition of Ambiguity

Recognition of ambiguity in the problem specification was determined (from transcripts) based on whether students asked questions or made assumptions while doing the design. Although recognition of ambiguity is a recognized expert characteristic [4], FC students in this study were more likely to make (usually simplifying) assumptions than to question ambiguous requirements [2].

No statistically significant differences were found in the ability of students to recognize ambiguity based on either the early language of instruction or the paradigm of instruction. The data shows that 56% of C++ learners and 67% of Java learners recognized ambiguity. Furthermore 40% of functional learners, 54% of imperative learners and 70% of OO learners recognized ambiguity. This data reports on all 136 students.

4. DISCUSSION

4.1 Statistically Significant Results

Three statistically significant findings are reported in this work. First, C++ learners are more likely to produce hierarchical, nested, or grouping structure in their designs than are Java learners. This finding may be of interest to the education community as it implies that the C++ language alone (regardless of paradigm of instruction) encourages students to produce designs with organizational structure that we deem to have value.

The other two statistically significant results report on the subset of students who identify interaction between components in their design and relate to how many design requirements they *attempt* to address. Of this subset, C++ learners attempt to address more requirements than Java learners. Additionally, the subset of imperative learners who identify interaction attempt more requirements than OO learners. We expect these findings to be of less interest to the educational practitioner. Not only do these results concern only those students who identify interaction in their design (an "expert" characteristic), but the result refers to students' *attempts* to address requirements. The level of success of these attempts is not rated, and as such, is of limited use in helping define a change in educational practices based on this result.

4.2 Relative non-impact of early language and paradigm

Clearly this work found very few statistically significant differences between students based on early language or paradigm. One conclusion, then, would be that despite many educators' strong views on best practices in terms of first programming paradigm and/or language, there are ***very few*** differences in FC generated software designs that can be explained by differences in early programming paradigm or language. Of course this work does not comment on other reasons why a particular language or paradigm might be preferred. Nor does this work prove that there is no effect of early language or paradigm on students' ability to design. Rather, a reasonable conclusion might be that the experiment conducted, the data gathered, and/or the analysis performed were not sufficiently sensitive to catch any such effects.

Further work on the relationship between paradigm and design ability might focus on determining what types of differences are expected between OO learners and imperative learners. What specific effect do we expect a first programming course that uses a particular language or paradigm to have on student understanding of design? Can we develop a design task or other experimental instrument that would enable us to determine if a particular characteristic demonstrating this difference does, in fact, occur more frequently for one set of paradigm (or language) learners? Practically, any results would need to generalize across types of institutions and individual instructors, so can we define the characteristics in sufficient detail that multiple educators will agree whether a given design possesses it?

5. CONCLUSIONS

This work reports on the results of a design task undertaken by 136 first competency students in a multi-national, multi-institutional project. The design representations produced and the

transcripts of student think-alouds were probed for a variety of surface characteristics. These were then analyzed for differences due to early-language and early-paradigm instruction.

In general, the statistical tests revealed little in these characteristics when analyzed across students' early programming language of instruction and their early programming paradigm of instruction. Only three statistically significant differences were found, the most interesting of which showed that C++ learners are more likely to produce designs evidencing hierarchical, nested, or grouping structure than Java learners. Notably, that same distinction cannot be found between the imperative and OO paradigms. Hopefully this work sheds some light on the commonly perceived importance of particular first languages or paradigms of instruction and encourages deeper consideration of how, and whether, we believe first programming language and paradigm should (measurably) affect design ability.

6. ACKNOWLEDGMENTS

We would like to thank the Scaffolding (and Bootstrapping) members who commented on early versions of this paper and who gathered and analyzed data reported on here. This material is based upon work supported by the National Science Foundation under Grant No. DUE-0243242. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

7. REFERENCES

[1] Astrachan, O. L., Bruce, K. B., Koffman, E. B., Kolling, M., Reges, S. Resolved: objects early has failed. In *Proceedings of SIGCSE*. St. Louis, MO. March, 2005. 451-452.

[2] Blaha, K., Monge, A. E., Sanders, D., Simon, B., VanDeGrift, T. Do students recognize ambiguity in software design? A multi-national, multi-institutional report. In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*. 2005. 615-616

[3] Bruegge, B., Dutoit, A. Object-Oriented Software Engineering. Prentice Hall. 2000.

[4] Bursic, K. M., Atman, C. J. Information gathering: a critical step for quality in the design process. Quality Management Journal 4 (4). 1997. 60-75.

[5] Chen, T.-Y., Cooper, S., McCartney, R., Schwartzman, L. The (relative) importance of software design criteria. In *Proceedings of ITiCSE*. Monte da Caparica, Portugal. June, 2005. 34-38.

[6] Fincher, S., Petre, M., Tenenberg, J., Blaha, K., Bouvier, D., Chen, T.-Y., Chinn, D., Cooper, S., Eckerdal, A., Johnson, H., McCartney, R., Monge, A., Mostrom, J. E., Powers, K., Ratcliffe, M., Robins, A., Sanders, D., Schwartzman, L., Simon, B., Stoker, C., Tew, A. E., VanDeGrift, T. A multi-national, multi-institutional study of student-generated software designs. In *Proceedings of Kolin Kolistelut – Koli Calling*. 2004.

[7] Fix, V., Wiedenbeck, S., Scholtz, J. Mental representations of programs by novices and experts. In *Proceedings of Interchi '93*. April, 1993. 74-79.

[8] Hadjerrouit, S. Java as first programming language: a critical evaluation. SIGCSE Bulletin 30 (2). June 1998. 43-47.

[9] Hamilton, J. A., Murtagh, J. L., Zolier, R. G. Programming language impacts on learning. Ada Letters. September, 2000. 12-19.

[10] Jeffries, R., Turner, A. A., Polson, P. G., Atwood, M. E. The processes involved in designing software. In *Cognitive Skills and Their Acquisition*. Erlbaum. Hillsdale, NJ. 1981. 255-283.

[11] Lahtinen, E., Ala-Mutka, K., Jarvinen, H.-M. A study of the difficulties of novice programmers. In *Proceedings of ITiCSE'05*. June, 2005. 14-18.

[12] McCracken, W. M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B.-D., Laxer, C., Thomas, L., Utting, I., Wilusz, T. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. SIGCSE Bulletin 33(4), 2001. 125-140.

[13] McCracken, M., Newstetter, W., Chastine, J. Misconceptions of designing: A descriptive study. In *Proceedings of ITiCSE*. Cracow, Poland. June, 1999. 48-51.

[14] Or-bach, R., Lavy, I. Cognitive activities of abstraction in object orientation: an empirical study. SIGCSE Bulletin 36 (2). June, 2004. 82-86.

[15] Tenenberg, J., Fincher, S., Blaha, K., Bouvier, D., Chen, T.-Y., Chinn, D., Cooper, S., Eckerdal, A., Johnson, H., McCartney, R., Monge, A., Mostrom, J. E., Petre, M., Powers, K., Ratcliffe, M., Robins, A., Sanders, D., Schwartzman, L., Simon, B., Stoker, C., Tew, A. E., VanDeGrift, T. Students Designing Software: A Multi-National, Multi-Institutional Study. In Informatics in Education, 4(1). 2005. 143-162.

[16] Vandenberg, S., Wollowski, M. Introducing computer science using a breadth-first approach and functional programming. In *Proceedings of SIGCSE*. Austin, TX. March, 2000. 180-184.

[17] Weisert, C. Learning to program: it starts with procedural. *Information Disciplines, Inc*. June, 1997.