Evaluating Languages and Environments for Novice Programmers

Linda M^{c}Iver
School of Computer Science and Software Engineering
Monash University, Australia
linda.mciver@csse.monash.edu.au

Keywords: POP-I.B. choice of language, POP-II.A. novice programmers, POP-III.D. editors and debuggers

Abstract

Although debate rages strongly over which programming language is the best for any particular application (especially for teaching introductory programming), there is a lack of objective data informing the discussions. There is a similar lack of data on appropriate development environments for introductory programming courses. This paper discusses an existing method of comparing and evaluating programming languages, and how that method can be adapted to compare and evaluate integrated development environments, with a particular focus on environments for novice programmers.

Introduction

While there are many strong feelings on the subject of first programming languages, there have been few evaluative studies of languages, or development environments, for novice programmers. Anecdotal evidence from introductory programming courses abounds (Reed, 2001; Allen, Grant, & Smith, 1996; Popyack & Herrmann, 1993), and individual language features have been studied from a cognitive point of view (Soloway, Bonar, & Ehrlich, 1989; Sime, Green, & Guest, 1973), but the question of which programming language should be used for teaching introductory programming remains contentious. Suggestions range from Mail Merge as a first programming language (Popyack & Herrmann, 1993), through to traditional choices such as Pascal, scripting languages such as Javascript (Reed, 2001), and extremely complex, "industrial" programming languages such as Ada (Allen, Grant, & Smith, 1996) or C++.

The evaluation process for a programming language typically involves years of experience with the language in its intended environment, be it industrial software engineering, computer science education, research computing, or recreational programming (Allen, Grant, & Smith, 1996; Brusilovsky et al, 1994; Collins & Fung, 1999). Formal evaluation programmes are few and far between, and most evidence gathered is anecdotal in nature. In an educational setting, the demands of courses and curricula make it difficult, if not impossible, to compare different languages in the same course. Different courses generally have sufficiently different curricula to make language comparisons meaningless. Financial constraints limit opportunities for formal comparisons in industry. Where comparisons can be made between courses or projects, the number and scale of differences between the different settings frequently obscure the results.

It is has been more common to compare single attributes, for example language constructs, rather than whole languages (Soloway, Bonar, & Ehrlich, 1989; Sime, Green, & Guest, 1973). This approach is useful to the field of language design, as it gives firm indication of the value and impact of individual features, where comparison of entire languages does not easily lend itself to analysis of particular features within the languages. However, this technique leaves unanswered the question of which language is best for a particular task, and the interaction between language features is often neglected.

A stand-alone language evaluation has been done by Eisenstadt & Lewis (1992), who analysed a large number of errors made by students using the SOLO language. This analysis has informed the development of the evaluation techniques described here. The analysis of errors in SOLO was made with the aim of eradicating as many syntax errors as possible, by using the analysis to inform the modification of the interactive environment. A similar style of observation is also reported in Thomas & Paine (2000), where the focus is on developing an electronic coaching system for distance education, rather than on evaluation of the language or environment.

Usability evaluation of a complete programming language has been done by Clarke (2001), using the Cognitive Dimensions questionnaire (Blackwell & Green, 2001), but it is doubtful whether novice programmers would have the necessary skills and background knowledge to answer the questionnaire effectively.

There is some evidence that a well-designed programming environment can assist students learning to program (Eisenstadt & Lewis, 1996) but once again there have been few, if any, direct evaluations of whether the choice, or design, of programming development environment has a real impact on learning.

Development environments vary dramatically, from simple text editors and command-line compilers to fully interactive and integrated development environments. For the purposes of this paper, "environment" will be used to refer to integrated development environments, where the programmer interacts with one piece of software that provides editing, debugging, compilation/interpretation, and sometimes visualisation tools.

This paper describes a method being developed at Monash University for the empirical study of development environments, both comparatively and in isolation, in order to further our knowledge of both design and selection principles for integrated development environments for novice programmers. A case is made for the need to conduct both comparative and stand-alone evaluations to find the best development environment for any task, and to find out how much impact the development environment has on learning, productivity, and student satisfaction and frustration.

Evaluation of Languages

Comparative evaluation of programming languages can provide a great deal of information about the relative usability of each language. It can inform the debate about which programming language to use in introductory programming courses, or which language is best for doing particular types of tasks. It allows direct comparison of languages, and hence a form of benchmarking - programmers using language *A*, for example, might make 10% more errors on a specific type of task than programmers using language *B*. This sort of benchmarking would allow more objective and informed decisions to be made wherever choice of language is an issue. In addition to informing choice of language, comparative evaluations can inform the future design of programming languages, as they make more information available about the use of the complete language, not just individual constructs.

A method for comparative evaluation of programming languages, based on standard usability principles, is described in McIver (2000). In this method, programmer[1] interaction with the language is recorded, and the number and type of errors made is analysed. The environment is standardised, and simplified, so that both languages are accessed using a simple text editor with a large "RUN" button which compiles and either returns error messages or runs the code. In this way differences in environment are eliminated, and any differences in interaction are solely due to the different languages used in the trial.

Errors are divided into syntax and logic errors. Analysis can be taken further by breaking errors down into sub-categories by language construct involved. More time-consuming, and arguably most interesting, is to analyse the path taken when students are trying to correct errors and solve problems. How do they respond to the different types of errors? How long does it take them to correct their syntax errors? Their logic errors? etc.

This method is useful for comparing languages for a specific task, but it does have some drawbacks. The selection of programming problems is difficult to make language independent - some problems will naturally be more suited to one language than the other. Comparison of very different languages is thus difficult to make truly objective and fair. For very specific circumstances, however, such as introductory programming courses, or projects where specific types of problems will need to be solved, this method of comparison is quite appropriate, since the comparison is not absolute, but

[1] in this case, students learning programming for the first time

rather relative to the intended use of the language, and the intended users. Despite the strong views of some language evangelists, task- and user-specificity are the most meaningful contexts for language comparison - "is language A better than language B *for my specific purposes*".

The evaluation of programming languages leads naturally to the evaluation of different development environments for a single programming language, since the development environment can have a significant impact on the overall usability of a programming language (Kölling & Rosenberg, 1996). Feedback from the compiler, the presence or absence of syntax-directed editing, help systems, and visualisation tools all affect a programmer's interaction with a programming language.

The method of evaluation of programming languages described above is easily modified to evaluate environments as well - and it can also be modified to allow stand-alone evaluation of a single language or environment, rather than comparative evaluation. These modifications are described in the following sections.

Evaluation of Environments (Comparative)

The comparative evaluation of programming environments for novice programmers is similar to that for programming languages, with the advantage that learning of the language can be measured (through the use of pre- and post-tests). This is much harder with comparative evaluation of programming languages, because programming knowledge is notoriously difficult to measure in a language-independent fashion, especially at an introductory level.

As with language comparison, the evaluation involves taking two different development environments (for the same programming language) and comparing programmer interaction with them. Errors made while using the environment are recorded (and language errors can also be recorded, to see if the environment has a direct impact on language use), as well as which parts of the environment actually get used by programmers. Do they use the debugging facilities? Do they access online help, and if so, does it help them? Which menu options do they use? Do they use features the way developers envisioned?

Quantitative data can be collected for, among other things, numbers of errors, types of errors, time-to-completion for various tasks, and time spent using different parts of the environment (for example, do programmers spend most of their time editing the code? How long do they spend using visualisation tools, or debugging tools?).

As well as measuring interaction, performance metrics can also be applied - in the case of students learning programming, pre- and post-tests can be used to measure learning and understanding, while in the case of professional programmers, standard performance metrics for productivity, efficiency, and accuracy can be used.

Qualitative data can be obtained by tracing the paths taken by students attempting to solve problems - for example, if a student receives an error message from the system, what does she do next? If a student receives an unexpected result from a program, which part of the system does he turn to in his attempt to trace the problem? This sort of data is expensive to analyse, since it is mostly not amenable to automatic processing. Hence it is not done for every student, or even a large number of students. In most cases a sample of interesting errors is selected, and the data traced through by hand for each one. However, interesting results can be obtained using this technique.

Similarly, the data can be analysed for trends in terms of the way students set out to tackle a problem. For example, if the environment offers a visualisation tool, at which point in the development cycling (if any) do students typically access this tool? Do they use it for designing the program, or for debugging, or for checking and validating their solutions?

Depending on when the evaluation is performed (eg. at the beginning of the first semester of an introductory course, or in the middle, or towards the end), different results will be obtained. Environments which provide a wide range of tools are usually introduced gradually to students, so that the whole environment may not be used, regardless of how well-designed it is. For this reason, the evaluation must take the course syllabus and timing into account. Different evaluation objectives will be better suited to different parts of the course - for example, evaluating the learning curve at the start of semester, or evaluating the use of the tool in a reasonably large and complex assignment.

Having detailed information about the comparative merits of different environments will allow informed, objective choices to be made. Environments which show clear evidence of good programmer support – in that errors are more quickly resolved, or perhaps even less common to start with – are clearly a better choice for introductory programming courses than environments with higher error rates, consistently misleading online help or compiler error messages, or debugging facilities with steep learning curves.

In addition to specific information about particular development environments, this style of evaluation has the potential to give valuable insights into programming in general, and supportive environments in particular. Such information could be used to inform the future design of development environments, as well as the design of programming languages themselves. It has the potential to generate large amounts of data, some of which can be automatically processed, and it allows objective comparison of programming environments, together with details on the advantages and disadvantages of individual systems.

This style of comparative evaluation is difficult to apply in a course environment, due to the ethical and practical problems which arise from providing students in the same course with different facilities. Evaluating environments used in different courses is difficult, due to the large number of variables between different courses - different teaching styles, different syllabi, and different background among the students (for example, percentage of students who have programmed before, and what degree students are enrolled in, and what major they intend to pursue). Running small trials purely for comparative purposes (as opposed to gathering data in situations where programming is already taking place, such as introductory programming courses or programming projects in industry) is expensive and time-consuming, and does not produce the comprehensive amounts of data for automatic processing that can be gleaned from a larger, in-situ, comparative study.

In addition, much of the data gathered in this type of comparison is just as useful in a stand-alone evaluation of a single environment. Converting the comparative evaluation process into a stand-alone evaluation is simple, and can be extremely informative. The necessary changes to the evaluation process are described in the next section, which is followed by a description of a planned application of a stand-alone evaluation.

Evaluation of Environments (in isolation)

Stand-alone evaluation of a single development environment is often more practical to apply than full comparative evaluation. Some of the same types of data can be collected, and, even without comparison, can yield important information about an environment's strengths and weaknesses.

The data gathered in a stand-alone evaluation can also be used to make inferences about student learning, and about problems with the programming language, as well as with the environment. An important aspect of this style of evaluation is the use of a speak-aloud protocol with a small number of participants, so that inferences about the data can be tested against what the students were trying to do, and how they were reasoning about the system. In this protocol, students execute a controlled task while describing what they are doing. Their comments and descriptions are recorded, and evaluators can also question students about why they make particular decisions, or use particular methods. As Eisenstadt and Lewis (1996) point out, *"Symptom and cause are not the same thing."* Knowing what the students did does not mean knowing why they did it. By using the speak-aloud protocol with some students, the validity of inferences about the causes of errors can be tested, and other information can be gathered about the use of the system - for example, if students did not use the debugger in the environment, it may be because the debugger was confusing, or had not yet been introduced.

Stand-alone evaluations have a number of advantages over comparative evaluations. They can be done with existing groups of students, and data can be automatically collected, without any extra effort from the students beyond the programming they were already doing as part of their course. Only a small group of students (those participating in the speak-aloud protocol) would need to commit extra time to the study. This obviates the need for setting up expensive, special-purpose trials purely for data collection, and avoids the ethical problems inherent in trying to provide students within the same course with different materials and facilities.

Stand-alone evaluation, using a large group of students, can generate a very large amount of data, much of which can be automatically processed, and used to provide a detailed picture of students' interaction with the system. While some of the data cannot be as readily processed automatically, qualitative evaluation can be done using a reduced sample of work, and where interesting results arise, that sample can easily be extended.

Given these large amounts of data, some comparison (albeit limited and general comparison) can be done with other evaluations, even across courses. While the same detailed comparisons cannot be done as in a true, comparative evaluation, some inferences could still be drawn from different data sets. Figures which could still be compared include error rates, and student response to the various error messages, online help systems, etc. These comparisons are not strictly valid, since other factors in the courses could impact on the figures, but they may provide pointers to important differences between the environments.

Eventually, given a sufficiently large range of evaluations conducted in different courses, it would be extremely interesting to compare the results, especially if the same environment is evaluated in multiple settings. This is discussed further in the next section, which describes a forthcoming application of the stand-alone evaluation technique.

In addition to providing information about the programming language and development environment, an evaluation of this kind would directly inform the teaching of the course involved, by providing large amounts of data on which parts of the language and environment students understand and use effectively, and which they don't. It would also provide a detailed picture of overall student interaction with the language, and show patterns of use for the entire student cohort that are rarely available.

Results showing detailed information about the usability of the environment, and students' responses to things like the debugging environment, the error messages, the online help, etc, can also be used to inform and improve the ongoing development of the environment. Error messages and online help can be improved, problem areas of the interface can be redesigned, and usability issues can be resolved - all in an educated fashion, rather than based on purely anecdotal evidence.

One of the disadvantages of stand-alone evaluation is that no specific comparison with other studies is possible for impact on learning, since course, teaching, and student differences are likely to have a large impact on learning, thus obscuring the impact of the development environment.

This means that stand-alone evaluation does not readily inform the "which environment" debate, because it does not allow for objective "benchmarking" of different environments. Nonetheless, as shown above, it does provide considerable information to teachers, students, and the designers of development environments.

Both stand-alone and comparative evaluation will be more difficult when the environment to be evaluated is a commercial tool, or a tool for which the source code is not available, since the simplest way to collect the data is to modify the environment to collect most of the data automatically. Other techniques can be used, including video tapes of programmer interaction, or programs that record everything that takes place on a machine, but they all require substantially more work to process and analyse the data, making the evaluation of commercial environments, in general, prohibitively expensive and resource-intensive.

Applied Evaluation

While the evaluation of programming languages has been applied with considerable success (M^{c}Iver, 2001), the evaluation of development environments has yet to be applied. The first trial of this evaluation method will be an evaluation of BlueJ (Kölling & Rosenberg, 1996). BlueJ is a Java environment that combines syntax-directed code editing with visualisation capabilities, compilation, debugging, and the ability to execute the code.

An important part of the development of this evaluation process will be the pilot study run with BlueJ, before the full evaluation takes place. The pilot study has two primary objectives. Firstly it will be used to determine what sort of data can readily be collected, and how it can be analysed. Secondly, the speak-aloud protocol will be used to test inferences about the meaning of the behaviour observed during the pilot.

A researcher, who was not present during the session where the speak-aloud protocol was used, will analyse the data and make inferences about what students were trying to do during the session. This will include inferences about errors students made, and why they tried particular ways of solving them, as well as inferences about the way students use the system. These inferences will then be compared against what students actually said while they were working, so that some measure can be obtained of how valid such inferences are. A range of students will be selected for this part of the evaluation based on achievement. As the pilot study will be run in the second semester of the introductory programming course, students will be chosen from several bands of achievement in the first semester subject: 50-60%, 60-70%, 70-80%, and 80-100%.

BlueJ is currently used in over 150 institutions around the world for teaching Java programming (www.bluej.org). Because of BlueJ's popularity, this evaluation can eventually be extended to numerous institutions in many different countries. This will provide more data on BlueJ, and it will also allow data from different courses to be compared in order to test the validity of comparisons across courses. If the BlueJ data is largely the same across different institutions running different courses, this may indicate that different environments can be meaningfully evaluated using different courses. This would eliminate one of the problems with obtaining comparative data on development environments used for teaching. On the other hand, if the BlueJ data is significantly different across different courses, that provides strong evidence for the questionable validity of the comparative evaluation of programming environments in different courses.

It would be particularly interesting to examine the results from significantly different teaching methods, such as face-to-face and distance education, to see if students in these different courses have substantially different interaction with the programming environment.

The evaluation of BlueJ will not initially be comparative, although the data can readily be extended to a comparative study given time, funds, and appropriate sources of data.

Conclusions

Evaluation of programming languages and development environ-ments can provide important information for the design of new systems, and the selection of existing ones. The development of a consistent evaluation technique may allow comparative evaluation across different courses, businesses, and even countries, so that more information can be collected to inform the contentious debate on which programming language and environment is best suited to a particular task, and a particular type of user. The models presented here for comparative and stand-alone evaluations are a first step towards the collection of data on many different programming languages and environments.

References

Allen, R.K., Grant, Douglas D., & Smith, R. (1996) Using Ada as the first Programming language: A Retrospective. In *Proceedings of Software Engineering: Education & Practice, 1996 (SE:E&P'96)*, IEEE Computer Society Press.

Blackwell, A.F. & Green, T.R.G. (2000) A Cognitive Dimensions questionnaire optimised for users. In A. F. Blackwell & E. Bilotta (Eds.) *Proceedings Twelfth Annual Meeting of the Psychology of Programming Interest Group*, Corigliano Calabro, Italy, April, Edizioni Memoria.

Brusilovsky, P., Calabrese, E., Hvorecky, E., Kouchnirenko, A., & Miller, P. (1997) Mini-languages: A Way to Learn Programming Principles. In *Education and Information Technologies*, 2(1): 65-83.

Clarke, Steven (2001) Evaluating a new programming language. In G Kokoda (Ed.) *Proceedings of the 13th Annual Workshop of the Psychology of Programming Interest Group.*

Collins, Trevor D., & Fung, Pat (1999) Cognitive Modelling for Psychology Students: The Evaluation of a Pragmatic Approach to Computer Programming for Non-Programmers. In G Cummings, T Okamoto and Louis Gomez (eds), *Proceedings of the 7th International Conference on Computers in Education (ICCE'99)*, Chiba, Japan, November, IOS Press, Volume 1, pp 216-223.

Eisenstadt, Marc and Lewis, Matthew W. (1992) Errors in an Interactive Programming Environment: Causes and Cures. In *Novice Programming Environments: Explorations in Human-Computer Interaction and Artificial Intelligence,* Marc Eisenstadt, Mark T. Keane, and Tim Rajan, (eds), Lawrence Erlbaum Associates, Hillsdale USA.

Kölling, M. and Rosenberg, J. (1996) An Object-Oriented Program Development Environment for the First Programming Course. *Proceedings of the 27th SIGCSE Technical Symposium on Computer Science Education*, ACM, Philadelphia, Pennsylvania, March, pp.83-87.

M^{c}Iver, L. (2001) Syntactic and Semantic Issues in Introductory Programming Education. *PhD Thesis. Monash University*, Melbourne, Australia. (available on the web : http://www.csse.monash.edu.au/~lindap/papers/LindaMcIverThesis.pdf)

M^{c}Iver, L. (2000) The Effect of Programming Language on Error Rates of Novice Programmers. In A. F. Blackwell & E. Bilotta (Eds.) *Proceedings Twelfth Annual Meeting of the Psychology of Programming Interest Group*, Corigliano Calabro, Italy, April, Edizioni Memoria.

Popyack, J.L. & Herrmann, N. (1993) Mail merge as a first programming language. In Klein BJ (ed) *Twenty-Fourth SIGCSE Technical Symposium on Computer Science Education*, Association for Computing Machinery Special Interest Group on Computer Science Education, Grand Valley State University, MI, pp. 136-140.

Reed, David (2001) Rethinking CS0 with JavaScript. *Proceedings of the 32nd SIGCSE Technical Symposium on Computer Science Education*, SIGCSE Bulletin 33(1), 2001

Sime, M. E., Green, T.R.G., & Guest, D.J. (1973) Psychological Evaluation of Two Conditional Constructions Used in Computer Languages. *International Journal of Man-Machine Studies*, 5, pp 105-113.

Soloway, Elliot, Bonar, Jeffrey & Ehrlich, Kate (1989) Cognitive Strategies and Looping Constructs: An Empirical Study. In E. Soloway and J.C. Spohrer, editors, *Studying the Novice Programmer*, Lawrence Erlbaum Associates, Hillsdale.

Thomas, P.G., and Paine, C. B., (2000) Tools for Observing Study Behaviour. In A. F. Blackwell & E. Bilotta (Eds.) *Proceedings Twelfth Annual Meeting of the Psychology of Programming Interest Group*, Corigliano Calabro, Italy, April, Edizioni Memoria.