Learning Curve Management in Educational Programming Environments

Benjamin H. Brinckerhoff
Computer Science and Engineering
Washington University
St. Louis, MO
bhbrinckerhoff@wustl.edu

Kenneth J. Goldman
Computer Science and Engineering
Washington University
St. Louis, MO
kjg@cse.wustl.edu

ABSTRACT

Beginning programmers are best served by integrated development environments that adapt to their growing sophistication as programmers. To this end, we propose four design goals for learning curve management in educational programming environments. We provide pedagogical justification for each goal, describe possible supporting feature sets, and discuss the extent to which these goals have been achieved in some current environments, particularly JPie, our interactive environment for live construction of Java applications.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environments – *graphical environments, integrated environments, interactive environments;* K.3.2 [**Computers and Education**]: Computer and Information Science Education – *computer science education, curriculum;*

General Terms

Human Factors

Keywords

Dynamic classes, live programming.

1 INTRODUCTION

We define *educational programming environments* as integrated development environments (IDEs) that serve the needs of those who are learning how to create software. While professional programming environments can satisfy their audience with a relatively fixed set of features, we contend that educational programming environments (EPEs) must adapt to the needs of programmers who become more sophisticated over time. Beginning programmers require features that help them learn the programming model through the construction of simple programs, whereas more experienced programmers require features that assist with writing robust software. The traditional route is for programmers to switch to a different programming environment at various stages in their development, but this happens infrequently due to the relatively high overhead of making that transition. Moreover, the level of programmer sophistication changes most rapidly at the beginning of a student's career, when disruptions due to a change in programming environment would do more harm than good. Therefore, an ideal educational programming environment provides *learning curve management* features to allow the student, the instructor, or even the environment itself, to control the way the IDE interacts with the programmer over time as the he or she gradually becomes more sophisticated.

In this paper, we present four design goals for learning curve management in EPEs. We discuss the importance of each goal and suggest specific features that can be implemented to achieve them. For illustration, we draw examples from JPie [3], DrJava [1] and BlueJ [5].

2 DESIGN GOALS

We propose that learning curve management can be effectively addressed in EPEs by providing dynamically changing feature sets in the following four areas. Throughout the paper, we use the term *target language* to refer to the high level programming language that the environment is designed to support. In our examples, Java is the target language.

Managing Errors. Beginners spend significant time finding and fixing errors. EPEs can assist programmers by providing flexible and configurable error management features that help prevent, detect, and mitigate errors, as well as inform programmers about the nature of errors. Initially, these features may constrain editing operations to prevent beginners from committing common errors. More advanced programmers may edit in less restrictive modes and correct errors themselves.

Reducing Complexity. Learning a new language can be overwhelming. Beginners must contend with complicated syntax, strange constructs and massive standard libraries. Environments can simplify programming by offering simple representations of language constructs, explicitly displaying information on language features, and only revealing essential parts of the standard library API. When programmers are ready, environments can make the language and libraries available in their original, more complex form.

Streamlining Programming. Making programming easier and more productive encourages beginners to continue learning. To this end, EPEs can include common productivity features seen in IDEs and allow direct interaction with objects. A more aggressive approach to streamlining programming includes providing a modified version of the target language (referred to as a *dialect*) that contains fewer rules and adds additional constructs for common programming tasks. The differences can be removed as the programmer advances, so that ultimately the student can program directly in the target language, without the "crutches."

Transitioning Users. EPEs can help users advance by adapting to address their needs at each stage of development, from beginner to expert. To support the development of language skills that are applicable outside of a specific environment, EPEs can include features that help users transition to traditional environments, such as professional IDEs or ordinary text editors.

3 RELATED WORK

The general goals and features of EPEs have been discussed in the literature [4]. In this section, we examine the work that has been done on those aspects of programming environments that pertain to learning curve management.

Error management systems (especially "help" systems with advanced information about errors) have been identified as a requirement for EPEs [4]. While many environments give useful information after an error has occurred, those that understand the syntax of the target language can also detect and/or prevent errors during editing [1,2,7]. The inclusion of debuggers to mitigate the effects of errors is also important for developing programmers [1,4,5].

Visual representation has been recognized as a powerful way to reduce the complexity of programming [4]. For instance, BlueJ [5] provides graphical representations of class relationships and Alice [2] uses a visual scripting language for 3-D animation. Such simple representations can help beginners grasp important concepts more easily.

The benefits of flexible language subsets have also been discussed [4]. Developing programmers may customize environments to use different language subsets that include more or less functionality and/or restrictions. Existing environments have also provided features to streamline programming. DrJava [1] provides an interpreted interactive editing mode (or *interaction pane*) and BlueJ [5] allows programmer to directly manipulate objects.

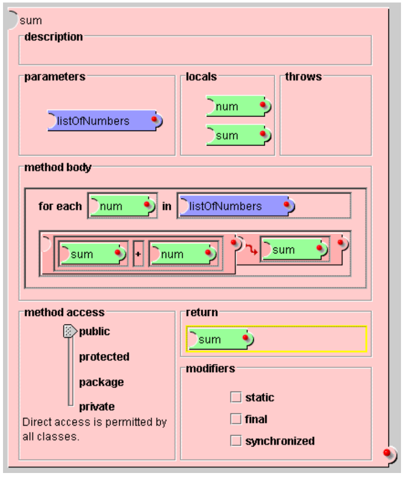
This paper considers these and other programming environment features in the broader context of learning curve management. The next four sections of the paper discuss in turn the four design goals stated in Section 2. We elaborate on the goals and describe ways in which they can be realized in programming environments. For illustration, we discuss the way these goals are approached in JPie [3], a visual programming environment we have developed for live construction of Java applications. Examples are also drawn from DrJava [1] and BlueJ [5].

4 MANAGING ERRORS

4.1 Preventing Errors

For beginners, who are unfamiliar with the target language, error prevention is preferable to error detection. Once an error is committed, beginners understand that an error exists, but may not know enough about the language to understand the error message and correct the problem. Environments that prevent errors altogether save beginners significant time and frustration while keeping them focused and motivated. Many programmers waste time concerned about syntax and compile-time errors instead of thinking about overall program structure and logic. By eliminating a common source of errors, EPEs can allow programmers to focus on more important conceptual issues [3].

One way to prevent syntax errors is for the environment to operate on a principle we call *gestural atomicity*, in which each user gesture takes the program description from one syntactically legal state to another, rendering "intermediate" illegal states impossible. For example, variable declaration in JPie is accomplished by placing a type into a scope as a single drag-and-drop operation. Similarly, as seen in Figure 1, control constructs are represented by graphical *templates* [7] that contain placeholders in which users can insert other templates, by selection or drag-and-drop.



Direct access is permitted by all classes.

Figure 1 - An example method in JPie.

By default, all programming constructs are represented graphically, allowing JPie to constrain editing and program construction to prevent syntactic errors and other errors such as undeclared variables and duplicate identifiers in the same scope. In general, each user action (drag-and-drop, keystroke, or selection) respects gestural atomicity. However, two kinds of syntactic errors can exist temporarily during program construction: (1) type incompatibility can occur along the way to forming a type safe statement or expression, and (2) empty expressions can exist in templates prior to their being filled in. The Alice system [2] uses gestural atomicity in the creation of method calls, forcing programmers to fill in the required parameters before moving on to other statements.

However, gestural atomicity is not always ideal for intermediate or advanced programmers. As a programmer becomes more familiar with the target language, dragging and dropping into templates can slow down the programming process and interfere with the programmer's train of thought. As such, programmers gradually come to prefer a less restrictive non-templatized editing environment in which they can directly type code statements and expressions, even if individual keystrokes may temporarily make the program ambiguous or syntactically incorrect. However, developing programmers may occasionally want to use templates, particularly when learning new constructs. To this end, JPie offers "mixed mode editing" that lets programmers switch back and forth between a graphical and textual representation of statements and expressions. Where possible, textual editing is still constrained (to prevent errors referencing non-existent class members, for example) yet retains the fluency of typing.

4.2 Detecting Errors

Some errors that are not prevented may be detectable by the environment. When an error is detected, the environment can alert the programmer using colored boxes and backgrounds, tooltip boxes, changed text font, size, or even animation.

All programmers benefit from immediate knowledge of errors, provided that notification is unobtrusive. Experienced programmers can note their errors and then either fix them immediately (while their focus is still on the incorrect part of the program) or ignore them (if the errors are expected and temporary). Beginners can immediately examine the error, realize their mistake, and avoid compounding their mistake.

However, beginners can also become overwhelmed by too many error messages. Related features such as error prevention (Section 4.1) and custom dialects (Section 6.3) can be used to automatically correct errors (explicitly or behind the scenes) for beginners. As the programmer develops, these features can become more "picky" as they correct fewer errors for the programmer. For example, JPie supports beginner programmers by automatically casting between many common types and does not require that all programmers catch all exceptions that could be thrown by a method call.

4.3 Mitigating Errors

Run-time errors such as infinite loops and recursion, deadlocks, and thrown exceptions cannot be detected by static analysis. However, EPEs can mitigate the impact of these run-time errors so they do not become major obstacles.

BlueJ helps users identify logic errors and infinite loops through a user-friendly debugger that allows manual halting and variable inspection at any point in program execution. DrJava's debugger allows variable inspection and interaction with running programs while the program is suspended.

JPie provides both loop and stack bounding to prevent infinite loops and infinite recursion. These bounds are set to default levels for beginners, but future versions of JPie will be manually configurable for more advanced programmers. When the bounds are reached, a dialog box appears, allowing the user to set larger bounds or to stop the execution. Although more advanced programmers can configure or disable this feature, the default bounds are welcome heuristics for beginning programmers.

Mitigating deadlocks can save significant time in debugging programs. For example, when JPie detects a deadlock, it a displays a resource allocation graph showing the cycle and gives the user an opportunity to open the offending threads in the debugger to see where they are blocked. The user may also break the deadlock by selectively aborting threads.

Thrown exceptions can also be mitigated. When an unhandled exception occurs (JPie does not require the programmer to catch or throw all exceptions within a method), JPie prevents the entire program from crashing by automatically launching an integrated debugger to show the source of the exception. Since JPie supports live modification, the programmer can correct the error or catch the exception and continue execution.

Beginning programmers benefit the most from these features. In the event of a program hang, an intermediate or advanced programmer may be able to locate the source, while a beginner may spend much more time finding the same error. But error mitigation is also useful for more experienced programmers since pinpointing the source of the error reduces their debugging time. When available, live program modification complements error mitigation, allowing programmers to fix errors on the fly and resume execution.

4.4 Explaining Errors

Whether an error is prevented, detected, or mitigated, the environment should tell the programmer what the error means and, ideally, how the user can correct or avoid it. To this end, EPEs generally include an integrated help system, easily accessed documentation and visual information.

Although the nature of the errors depends upon programmer sophistication, explaining errors is important for virtually all programmers. When an error is prevented, information about why the edit is illegal will avoid user confusion. When an error is detected, environments can provide information about how to correct the error. All error mitigation features can be accompanied by detailed information explaining why the run-time error occurred and how it can be avoided. Furthermore, runtime information facilities, such as debuggers and heap inspectors can help programmers apply the general information to their own specific instance of the error.

Information about errors is integrated into JPie. For instance, when a type error occurs in JPie, the problematic expression is highlighted and a mouseover action reveals information about the actual and expected type. An integrated help system gives the programmer more detailed information about the nature of errors and easily visible status bar give the programmer immediate feedback on errors. Integrated access to library documentation (javadoc) for method calls helps users avoid semantic errors such as passing the wrong parameter values.

Useful, easily accessible information allows programmers to advance their understanding of errors and avoid committing them in the future. While such information should be easily accessible, it should not be forced on programmers to the point of disruption. For instance, beginning programmers may ignore information about automatically prevented errors (see Section 4.1) due to an initial lack of language expertise, but a more sophisticated programmer may want to know about the error and why the system "fixed" things for them. Similarly, advanced programmers may intentionally temporarily commit some types of errors while constructing a program, and would not want to be prevented from doing so by an intrusive error reporting system. In any case, error explanations should ideally be tailored to the sophistication of the programmer, using only concepts and vocabulary familiar to the programmer.

5 REDUCING COMPLEXITY

5.1 Simple Representation

Learning to program is difficult. Programmers must simultaneously learn a new way of approaching problems and master a complex syntax and semantics. EPEs can make it easier by abstracting away the overwhelming details of the target language in order to reduce the mental overhead required by the student.

Visual programming environments can offer graphical representation and direct manipulation of constructs in the target language. By default, JPie represents all programming constructs graphically. After programmers master general programming concepts, they can tackle the more complicated textual representations of program constructs by switching into the textual editing mode. Similarly, BlueJ's visual representation of class relationships helps simplify complex ideas like inheritance and encapsulation.

5.2 Explicit Information

Programmers must not only remember the rules by which they can construct programs, but also what constructs are available. When declaring a method, programmers must remember which modifiers are available and which are appropriate. In Java, they could declare a method "abstract", "final", "static", or "synchronized". Access modifiers like "public", "private", and "protected" complicate matters further. Especially confusing are implicit modifiers like the implicit "package" modifier in Java.

Programmers must also remember the types of fields and methods. Programming languages by themselves give no clue as to the type of a field when it is being used or a method when it is being called. This is especially problematic in larger programs, where declaration and use may be spatially separated.

Environments can help minimize reliance on memory. For example, expression types in JPie are shown iconically and can be viewed textually by a mouse-over action. Modifiers appear as a series of checkboxes and along a slider when a field or method is being declared (Figure 1).

5.3 API Filtering

Many programming languages offer vast libraries containing hundreds of classes, each of which has many fields and methods. It can be nearly impossible for a beginner to determine which classes and methods are relevant to the problem at hand. This overwhelming level of complexity has been identified as a serious hindrance to effectively teaching computer science [6]. To solve this problem, an instructor can use API filtering to temporarily highlight useful classes in packages and hide unnecessary methods and fields from classes.

Java offers a huge number of classes in its standard library. To help students focus on the most relevant classes, JPie provides editable shortcut panels that can be grouped into categories and optionally loaded from a file created by an instructor. A new programmer might choose to use only the provided shortcuts, while more advanced students might explore the libraries and add their own shortcuts. In any case, the instructor can augment the set of shortcuts over time as students are exposed to more packages and classes.

An API filtering feature currently under development for JPie will allow instructors to hide irrelevant methods and fields in selected classes. When these *API filters* are loaded for specific classes, only those fields and methods deemed important will be visible to the user. Advanced programmers will be able to disable filtering to see all available fields and methods.

6 STREAMLINING PROGRAMMING

6.1 Facilitating Common Tasks

Many features in existing environments enable users to complete common programming tasks more easily, including automatic "get" and "set" method creation, a drag-and-drop GUI builder, and event recording, all of which JPie supports. DrJava provides syntax highlighting, automatic indentation, and bracket matching. Such features are useful for all programmers.

6.2 Direct Interaction

In object-oriented target languages, direct, fine-grain interaction with objects helps students understand the computational model and encourages them to experiment by quickly testing parts of their programs. For example, DrJava's interactions pane lets programmers evaluate Java expressions on the fly. Programmers can quickly experiment with code while avoiding the write/compile/run loop. BlueJ and JPie let programmers inspect the state of objects in the heap and call methods directly on those objects without running the entire application.

6.3 Custom Dialect

In most programming environments, programmers write directly in the target language. However, EPEs can help beginners learn by allowing them to construct programs in a *dialect*, or specialized version of the target language that makes programming faster and easier.

Dialects streamline programming by ignoring some of the rules of the target language so that programmers can quickly build programs without needing to anticipate every possible problem. Errors are then handled at run-time if and when they occur. A dialect can also offer programmers additional constructs that let them express common but complicated ideas simply and easily.

6.3.1 Fewer Rules

One limitation of programming languages, from an educational perspective, is that their rules are static and inflexible. A new programmer faces an imposing task: they must create working programs that strictly adhere to every single rule of the language, many of which he or she does not yet understand.

This "all or nothing" problem can manifest itself in several ways. Users may limit themselves to very simple applications, since these are the only ones that can be implemented without knowing more advanced features of the language. Ironically, interesting and useful applications are exactly those that motivate new programmers and help them appreciate they power of computing. An even more troubling problem occurs when students who are attempting to master a subset of the language encounter program errors that relate to aspects of the language they have not yet learned. For instance, users may want to pass a variable of type double to a method that expects a parameter of type int. If the user has not yet learned about casting, he is effectively barred from using that method. Users may become frustrated and stop trying to use new approaches to solve problems.

By using an educational environment with a custom dialect, programmers can ignore some rules of the target language in order to make programming easier. By ignoring some of these rules, programmers can create more interesting programs earlier on, and can be more confident with experimenting in programs, since they will always be using a familiar subset of the language. As the student advances, the dialect may be changed to force them to adhere to more and more rules. This promotes a natural learning process in which a small piece of the solution space is first understood and then slowly expanded.

The JPie Java dialect is more flexible than Java regarding types. JPie automatically coerces types as much as possible for the user, including automatic narrowing conversions and conversions to and from booleans and Strings. The JPie dialect also does not require programmers to catch or throw any exceptions. If an exception is thrown during execution, an integrated debugger is launched and asks the user to correct the problem at that point. In the future, more advanced programmers will be able to configure the environment to strictly enforce type and exception rules, effectively bringing the dialect closer to the target language.

6.3.2 *Support for the Common Case*

Using a custom dialect can provide useful programming abstractions and constructs for the beginning programmer. This expands the programmer's capabilities without introducing as many advanced programming concepts. For instance, a JPie *behavior* describes a periodic action to be carried out in a separate thread. The behavior in Figure 2 periodically calls methods to move a ball and check boundary conditions until the game is over. The template provides for specification of the rate, which in this example depends upon the current score. Additional constructs in JPie's dialect include a "for each" loop (similar to the one now available in Java 1.5) and a "match" statement (a generalized "switch" statement in which cases need not be constants).

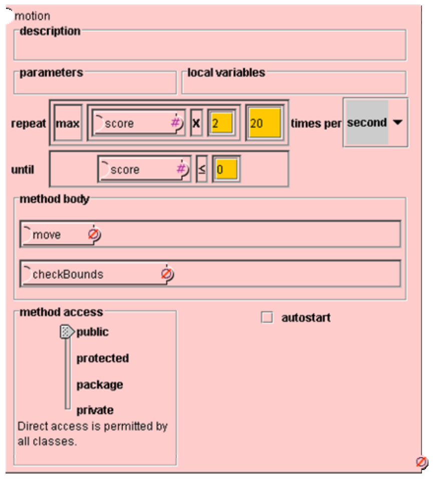


Figure 2 - A behavior in JPie.

7 TRANSITIONING USERS

7.1 Less-Restrictive Textual Editing

Since we hope that students will eventually need to program in an unrestricted textual programming environment (e.g. a professional IDE or text editor), we want EPEs to support this transition by including less-restrictive textual editing modes.

The more restrictive default editing mode (within JPie, this mode is also graphical) and the less restrictive textual editing mode can be used simultaneously, with some statements edited in the restrictive mode while others are edited in the unrestrictive mode. As a result, programmers don't need to master unrestrictive textual programming in one large step: rather, they may learn to textually edit the most familiar constructs first, and gradually add the rest, eventually moving to a completely unrestricted editor with the freedom to commit errors.

7.2 Source Code Generation

Automatically generated source code can help intermediate programmers learn the target language. Programmers can compare the generated code to the graphical representation (or dialect code) they have created to see their own ideas expressed directly in the target language Furthermore, they can experiment with editing generated code in order to gain hands-on experience with the target language.

Generated code should meet two criteria: it must be correct and it must be understandable. Clearly, the generated target language code must behave identically to the dialect program in the environment. To be educationally useful, the generated code must also be similar to the dialect code, demonstrate good programming style, and be simple enough that intermediate students can understand it.

8 CONCLUSION

We have advocated learning curve management as a means to address the changing needs of developing programmers. Instead of offering beginning programmers tools with static feature sets designed for experts, learning curve management can provide the basis for adaptive tools that provide appropriate levels of support for programmers as they develop from beginner to expert.

ACKNOWLEDGMENTS

We thank all the members of the JPie development team. This research was supported in part by the National Science Foundation under CISE Educational Innovation grant 0305954.

REFERENCES

[1] Allen, E., Cartwright, R., and Stoler, B. DrJava: A lightweight pedagogic environment for Java. *ACM SIGCSE Bulletin.*, 34, 1 (Mar. 1993), 137-141.

[2] Cooper, C., Dann, W., and Pausch, P. Teaching objects-first in introductory computer science. *ACM SIGCSE Bulletin.*, 35, 1 (Jan. 2003), 191-195.

[3] Goldman, K. An interactive environment for beginning Java programmers. *Science of Computer Programming, Special Issue on Practice and Experience with Java in Education*, Elsevier, to appear. Available at http://jpie.cse.wustl.edu/sub_sections/publications/scp.htm

[4] Jimenez-Peris, R., Pareja, C., Patino-Martinez, M., & Velazques, J. Towards truly Educational Programming Environments. Chapter from *Computer Science Education in the 21st Century*. Springer, 2000. 81-112.

[5] Kölling, M. & Rosenberg, J., Guidelines for teaching object orientation with Java. *In Proceedings of the 6th annual conference on Innovation and Technology in Computer Science Education* (Canterbury, England, June, 2001), 33-36.

[6] Roberts, E. The dream of a common language: the search for simplicity and stability in computer science education. In *Proceedings of the 35th SIGCSE technical symposium on Computer science education* (SCE '04) (Norfolk, Virginia, USA, March 3-7, 2004). ACM Press, New York, NY, 2004. 115-119.

[7] Tim Teitelbaum , Thomas Reps, The Cornell Program Synthesizer: a syntax-directed programming environment. Communications of the ACM, 24, 9 (Sep. 1981), 563-573.