Achieving Flexibility in Direct-Manipulation Programming Environments by Relaxing the Edit-Time Grammar

Benjamin E. Birnbaum and Kenneth J. Goldman
Computer Science and Engineering
Washington University in St. Louis
{beb2, kjg}@cse.wustl.edu

Abstract

Structured program editors can lower the entry barrier for beginning computer science students by preventing syntax errors. However, when editors force programs to be executable after every edit, a rigid development process results. We explore the use of a separate edit-time grammar that is more permissive than the runtime grammar. This helps achieve a balance between structured editing and flexibility, particularly in live development environments. JPie is a graphical programming environment that applies this separation to the live development of Java applications. We present the design goals for JPie's edit-time grammar and describe how its implementation supports a balance between structure and flexibility. As further illustration of the benefits of a relaxed edit-time grammar, we present "mixed-mode editing," an integration of textual and graphical editing for added flexibility.

1. Introduction

Learning syntax is a prerequisite to textual programming. However, it can distract beginners from deeper concepts, leaving them with the mistaken impression that computer science is about arcane rules instead of deep ideas.

Direct-manipulation programming environments (DMPEs) have been offered as a solution to this problem [8, 10]. In a DMPE, programs are presented graphically and can be manipulated with direct manipulation gestures like drag-and-drop. By only allowing gestures that lead to syntactically correct programs, a DMPE can prevent syntax errors, alleviating most syntax concerns for beginners.

However, if a DMPE requires that programs always be in an executable state, editing is less flexible. Therefore, there is a need for environments that find a balance between error-prone but flexible textual editing and safe but inflexible structured editing [10].

This paper presents an approach to adding some flexibility to DMPEs by separating the edit-time grammar (rules for valid programs after each edit) from the runtime grammar (rules for valid programs at execution). We argue that an edit-time grammar that is more permissive than the runtime grammar can provide the opportunity to control the balance between structure and flexibility.

A potential danger of using a relaxed edit-time grammar is that syntax concerns may resurface because programmers must fix the parts of the program that do not conform to the runtime grammar (called *discrepancies*) before execution can proceed. Therefore, it is important to choose an edit-time grammar carefully so that these discrepancies are few in number and easy to understand. Also, DMPEs that support live software development help alleviate this problem by allowing the user to handle discrepancies incrementally and as late as possible.

Even with a relaxed edit-time grammar, the graphical manipulation of programs may not be as quick or flexible as free-form textual editing. Beginners may benefit from the structure of graphical programming enough to warrant this inflexibility, but more experienced users may want a less restrictive environment that still provides more support than free-form textual editing. To this end, we present *mixed-mode editing*, the integration of flexible textual and graphical editing in an environment that prevents syntax mistakes. We show how a relaxed edit-time grammar makes it possible to integrate such a system of editing into a DMPE. That this form of editing can provide sufficient flexibility is supported by recent research showing that even experienced programmers

use only a small amount of the freedom provided by textual environments [11].

Our research vehicle is JPie [3, 4], a DMPE that uses separate edit-time and runtime grammars for flexible live development of Java applications.

We begin with an introduction to JPie and a discussion of related work. Section 2 presents principles for choosing an edit-time grammar and discusses JPie's edit-time grammar in relation to these principles. In Section 3, we show how a live development environment can lessen the burden of fixing discrepancies from the runtime grammar. Section 4 discusses mixed-mode editing and shows how our implementation was facilitated by JPie's relaxed edit-time grammar. Section 5 provides a preliminary evaluation of JPie based on classroom observations, and Section 6 describes directions for future work.

1.1 Background on JPie

JPie is a graphical programming environment that supports live development of Java applications. In JPie, programs can be modified as they run, and class changes immediately affect all existing instances. Live development proceeds through the use of *dynamic classes* [6] that represent program structure and enable live interpreted execution.

In JPie, programs are presented using a visual representation based on the syntax and semantics of Java. Figure 1 shows a method from a student lab to create an animated character ("sprite") that runs away from other sprites that come within its "radar."

The principle visual unit in JPie is the *capsule*, which represents types, variable declarations, variable accesses, properties, methods, method calls, constructors, constructor calls, and can also encapsulate constants and expressions. Users create programs by manipulating capsules through direct manipulation gestures and by building expressions using a calculator-like interface. Programs are divided into *semantic regions* [5], and the result of each gesture depends on the semantic region in which it is performed.

Because JPie keeps a persistent model of the program as it is being developed, it has the ability to accept editing gestures only if they can lead to structurally correct programs. As we will discuss in detail, JPie's definition of "correct" at edit-time is less restrictive than its definition at runtime. This allows increased editing flexibility without sacrificing the support of a DMPE. It also allows the integration of structured textual expression editing.

1.2 Related Work

Computer science educators have long recognized the difficulty of learning syntax and its potential to distract students from deeper ideas. A detailed account of efforts to make computer science more accessible for beginners can be found in [9].

One line of research has focused on preventing syntax mistakes through programming environments. The Cornell Program Synthesizer [14] was one of the first "syntax-based editors." Instead of presenting programs as a series of lines, the Program Synthesizer presents programs based on the underlying language grammar. It prevents most syntax mistakes by only allowing large-scale editing through templates that are embedded with the language's syntax. For smaller scale structures like assignment statements, it allows free-form textual editing but alerts the user to any syntax mistakes when expressions are completed. Like JPie, the Program Synthesizer supports live

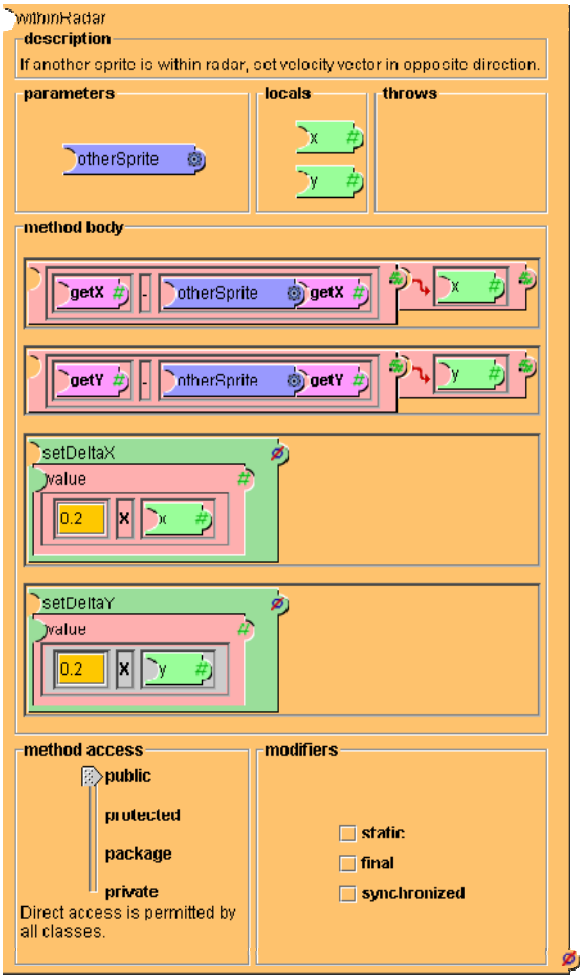

Figure 1. Graphical representation in JPie

development, in that programs can be executed until an empty statement is reached and then resumed after the statement is completed. The more flexible low-level editing resembles JPie's mixed-mode editing.

Direct-manipulation user interfaces [13] have enabled the transformation of the older syntax-directed editor concept into a graphical direct-manipulation programming environment (DMPE). These have been offered as a more promising means of using the programming environment to prevent syntax errors [12]. Like syntax-directed editors, DMPEs present programs based on structure instead of lines and can constrain edits so that only structurally valid programs are created. However, unlike syntax-directed editors, the structure of a DMPE is not necessarily based on the formal syntax of the underlying grammar. Instead, it may use a combination of syntax and semantics to represent programs. DMPEs get their name from their editing mechanism: programs are represented graphically and edited by direct manipulation of program components.

One of the most prominent DMPEs is Alice2 [2, 8], which introduces students to object-oriented programming through the creation of 3-D virtual worlds. Alice2 prevents all syntax mistakes through a drag-and-drop interface for creating programs. However, because the grammar that Alice2 enforces is the runtime grammar of the language, programs must be in an executable state after each edit. For example, if a method call is created, the expressions for each actual parameter must be filled in before any other edits can be made. Recent observations have shown that users of Alice2 were more likely to rewrite code than modify existing code [10].

In general, constrained visual editing of a complex language can be cumbersome. JPie's editing mechanisms differ from previous syntax-directed editors and DMPEs in that the grammar it enforces after each edit is more permissive than the runtime grammar. This provides more freedom in the order in which modifications to the program are made. In addition, JPie supports live program modification, which permits a more fluid development process and allows the programmer to incrementally modify portions of the program that do not conform to the runtime grammar. Furthermore, mixed-mode editing in JPie provides an alternative view that can help transition beginners to textual environments.

2. Relaxed Edit-Time Grammar

In this paper, we informally define a grammar to be a set of rules to which a program must conform. These rules are not necessarily the production rules of a formal grammar, but are the collection of predicates that a program must satisfy. The *edit-time grammar* is the set of rules enforced at each edit, and the *runtime grammar* is the set of rules that are required for execution. A *relaxed edit-time grammar* is an edit-time grammar whose rules are a proper subset of the rules of the runtime grammar. For each rule of the runtime grammar that is not present in the edit-time grammar, we say that there is a *relaxation* in the edit-time grammar. A *runtime discrepancy* (or *discrepancy*, for short) is a place in the program that conforms to the edit-time grammar but not the runtime grammar. Execution of the program cannot proceed past a discrepancy until it is fixed. For a given edit-time grammar, we can enumerate the types of discrepancies that can occur.

2.1 Design Goals for an Edit-Time Grammar

The purpose of a relaxed edit-time grammar is increased editing flexibility. For any given expression, there should be multiple ways of creating that expression. It should also be easy to change one expression to another through various editing paths.

However, the tradeoff to the added flexibility of a relaxed edit-time grammar is that beginners must use some of their conceptual resources to understand and correct runtime discrepancies. It is important to minimize this overhead. Live development of programs (described in detail in the next section) helps address this problem, but careful design of the edit-time grammar is crucial. Specifically, we offer the following principles:

- **Similarity** – The number of relaxations should be as small as possible. Each relaxation of the edit-time grammar results in one or more types of syntactic problems that are not prevented by the DMPE. These discrepancies may increase the conceptual load for programmers as they repair them. Therefore, relaxations should be made only if they significantly enhance flexibility.
- **Simplicity** – Only a basic knowledge of programming should be required to understand and correct each runtime discrepancy.
- **Error-locality** – Discrepancies should not require repairs that involve edits in more than one place. Error-locality makes discrepancies easier to conceptualize and repair.

2.2 JPie's Edit-Time Grammar

JPie's runtime grammar is modeled closely after Java's, although it contains a few modifications that

reduce complexity for beginning programmers (such as allowing uncaught checked exceptions). The edit-time grammar contains the following relaxations for increased editing flexibility:

- Empty expressions
- Type mismatches
- References to deleted methods and variables
- Empty operator placeholders

Both the first and the last of these relaxations facilitate the explicit representation and manipulation of non-terminals (defined in a formal grammatical sense) as if they were terminals. This allows the environment to preserve structure while supporting many editing patterns. In a flexible environment, the programmer must be able to temporarily leave a non-terminal incomplete while editing some other part of the program. For example, it may be desirable to use the value of a method invocation before filling in the actual parameter expression in that invocation. However, it is important that the DMPE explicitly represent the unfinished non-terminal so that the structure of the program is always apparent to the programmer. An example of this explicit representation in JPie is shown in Figure 2. Both the graphical and textual representations of a chain of method calls explicitly represent the actual parameter expression non-terminal as an empty box.

Runtime discrepancies in JPie are flagged at edit-time so that the programmer has the opportunity to fix them before execution. For example, empty expressions and type mismatches are shown with a red border and references to deleted methods and variables are grayed out.

Currently, JPie does not allow the user to customize the environment by deciding which relaxations are permitted, but one could imagine implementing such a feature in the context of learning curve management [1]. Limiting the relaxations for beginning users would lower the number of discrepancies they face at the expense of some flexibility.

In this section, we will show how the first three relaxations fit the design requirements from the previous section. (The last relaxation was made to increase flexibility in textual editing, so we defer its discussion to Section 4.2.3, where textual editing is discussed.) We will show that each relaxation significantly increases flexibility. Since there are only a few relaxations, the **similarity** requirement is satisfied. The other requirements will be discussed individually for each relaxation.

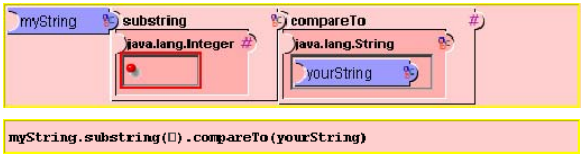

Figure 2. Representation of non-terminals

2.2.1 Empty Expressions

The first relaxation allows empty expressions to act as placeholders and to be manipulated as if they are not empty. These empty expressions can occur in many places, including an actual parameter expression of a method invocation, an operand of an arithmetic or boolean operator, the expression in a return statement, and the destination of an assignment statement.

Empty expressions permit increased abstraction and flexibility during editing by providing the freedom to create the skeleton of an implementation before filling in the details of each sub-expression. As concrete examples of this, consider the following scenarios that are made possible with empty expressions. A programmer can use a method invocation before deciding what the actual parameters should be; leave portions of arithmetic expressions not filled in while still manipulating them as if they are complete expressions; and create part of an assignment statement, realize that a local variable is needed to store the value of the assignment, and then fill in the rest of the assignment statement before creating the local variable. A more subtle advantage of allowing empty expressions is that formal parameters can be added to method declarations if there are already invocations of that method. Each existing invocation can gain an empty actual parameter expression for the new formal parameter.

In creating an edit-time grammar that facilitates this top-down expression building, there is an alternative to allowing empty expressions. When the programmer does not specify what an expression should be, the environment could supply a default value based on its expected type. This is the approach that Alice2 takes. However, we argue that this approach is inferior to simply allowing empty expressions. An expression that contains default values would look identical to an expression that happened to have those same values, which is undesirable because there would be no explicit way for the programmer to know that an expression with default values should be filled in. If users forget to fill in the correct values, they could face unexpected logic errors that are difficult to debug. JPie's solution of having empty statements that cause runtime discrepancies forces the user to fill them in before they are executed, avoiding such logic errors.

We have shown that empty expressions increase flexibility by allowing top-down editing patterns without forcing premature decisions about sub-expressions. The **simplicity** requirement is satisfied, since empty expressions are easy to understand, especially when represented explicitly. Finally, repairing an empty expression merely involves filling that expression, so **error-locality** is also satisfied.

2.2.2 Type Mismatches

In strongly-typed languages like Java, expressions have expected types. For example, the actual parameter expression has an expected type that matches the type of the formal parameter. JPie's edit-time grammar allows the type of an expression to differ from the expected type.

Allowing type mismatches provides the flexibility of incremental construction of expressions. For example, consider the construction of an actual parameter expression with an expected type of `Number`. An edit-time grammar that allows type mismatches permits the programmer to paste another expression that may have a type other than `Number`, say `Rectangle`. This is desirable if the programmer intends to call a method on this expression that returns a number, like `getWidth`. If, on the other hand, the edit-time grammar does not allow type mismatches, then the programmer must create the entire expression in one atomic editing step.

Type mismatches also increase editing flexibility by permitting programmers to change the type of a variable or the return type of a method, even if that variable or method is used elsewhere in the program. With type mismatches allowed, the types in expressions that access the variable or method can change along with the type of the variable or method itself. Any resulting type mismatches can simply be flagged and, if necessary, caught at runtime.

Types must be understood early by any user of a strongly-typed language, so type mismatches satisfy the **simplicity** requirement. A type mismatch merely requires an expression-level repair, so type mismatches also satisfy the **error-locality** requirement.

2.2.3 References to Deleted Methods and Variables

Allowing programs to contain references to deleted methods and variables provides the flexibility of deleting a declaration while there are still references to it and then replacing or deleting those references as appropriate. The alternative is to require that each reference to a method or variable be deleted before that method or variable is deleted, as required by Alice2. However, this increases the conceptual load for programmers, since they must think about what should happen to every use before deleting the declaration. This increase in effort may discourage making such edits and thus reduce flexibility. Moreover, if classes are edited in independent files, explicit representation of deleted items is necessary. Otherwise, one could never delete a public member because it might be referenced in another file that is currently not open.

Only a basic understanding of how method invocations and variable accesses work is required to understand why a reference to a deleted method or variable cannot execute, so the **simplicity** requirement is satisfied. The **error-locality** requirement is satisfied because each reference can be fixed independently.

2.2.4 Implementation

In JPie's backend representation, the nodes in the parse tree representing non-terminals have associated visual representations (textual and graphical) and have associated editors for replacing them by terminals or other expressions. After each edit, JPie traverses the affected subtree to provide error feedback for discrepancies. JPie uses the results of these validation tests during execution to pause the program at discrepancies and wait for the user to repair them.

3. Live Development

As discussed in the previous section, a risk of relaxing the edit-time grammar is that beginners must spend time fixing runtime discrepancies. We have discussed choosing an edit-time grammar to minimize the impact of discrepancies. We now show how live development in a DMPE lets users postpone decisions about discrepancies until runtime, extending the life of the edit-time grammar as long as possible into runtime and minimizing effort expended on discrepancies.

We define live development in a DMPE as the ability to edit programs as they run. If execution reaches a statement that has a runtime discrepancy, it can be paused until the statement is executable and then resumed from where it left off. Live development has a number of advantages that are beyond the scope of this paper [6], but its primary advantage in this context is that it eases the burden of fixing runtime discrepancies, thereby mitigating the extra attention to syntax that a relaxed edit-time grammar entails.

To understand how live development can ease the process of fixing discrepancies, consider the following example. Suppose that a user deletes a method in JPie. This action will cause all of its method calls to have the runtime discrepancy of a call to a deleted method. If the DMPE is a compiled environment, then the user may be forced to deal with each of these discrepancies before the program can run. However, these calls to deleted methods might be in sections of the program that are rarely or never executed. Furthermore, some of the calls might be in methods that are eventually deleted in their entirety. Postponing the resolution of these discrepancies until runtime can only decrease the number of discrepancies that must be resolved, since some of these discrepancies might never be executed. Even if all of the discrepancies are executed, dealing with them one at a time may simplify the process for the user. Thus, supporting live development can assist the user in using a relaxed edit-time grammar.

DMPEs lend themselves naturally to live development. To maintain syntactic integrity, a DMPE must maintain some backend representation of programs as they are edited. If this same representation is used for interpreted execution, then a change to this representation can immediately take effect on execution [6]. If a DMPE cannot support live development, then the discussion in Section 2 on editing flexibility still applies, but the user may be forced to fix all discrepancies before execution, which would make it harder for beginners to take advantage of a relaxed edit-time grammar.

4. Mixed-Mode Editing

To provide the flexibility of textual editing and to transition beginners to textual environments, a DMPE that integrates textual editing with direct manipulation is desirable. A DMPE that implements this *mixed-mode editing* must have a relaxed edit-time grammar, since intermediate states of textual editing are not always executable. In this section, we show how JPie's edit-time grammar supports this integration.

In JPie, mixed-mode editing is nested within the graphical view, as shown in Figure 3. For each expression, users can choose textual or graphical editing at will. Larger scale textual edits, such as control statements and blocks, are not currently supported. However, such support may not be necessary, since programmers rarely substitute one control construct by another [11].

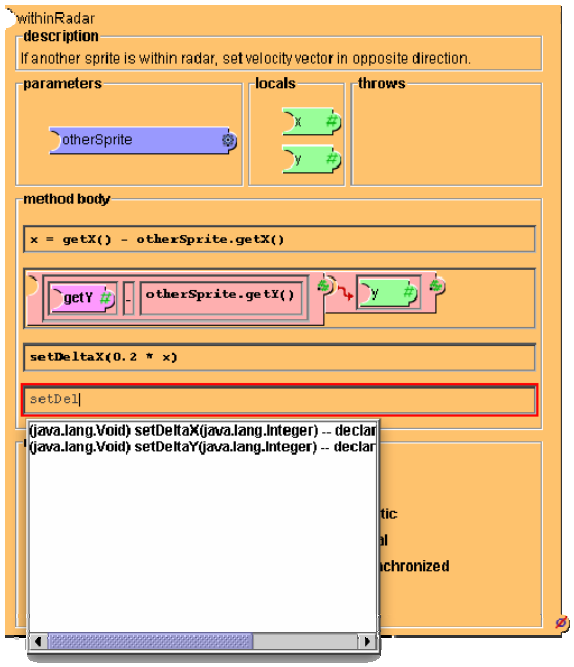

Figure 3. Mixed-mode editing in JPie

4.1 Requirements

The above ideas motivate the following design requirements for mixed-mode editing:

- **Syntactic Safety** – It should not be possible to type syntactically incorrect expressions. For example, if only `size` and `getSize()` are in scope, then typing `s` or `g` into an empty expression is allowed, but typing `d` is not, since it cannot lead to a syntactically correct expression.
- **Flexibility** – It should be possible to create an expression in a variety of ways, and these should be consistent with common editing patterns [11]. For example, it should be possible to add operators to an expression in a variety of orders.
- **Integration** – It should be possible to switch between graphical and textual editing and to mix direct manipulation gestures between the graphical and textual views.

In order to provide integration and syntactic safety, the program represented by the text must be consistent with the backend model of the DMPE. Therefore, the central challenge of mixed-mode editing is to keep the text as close to the model as possible without over-constraining the way programmers edit the text. Consistency and flexibility are competing concerns, and our goal is to find a careful balance between them.

4.2 Edit-time Grammar

The tradeoff between consistency and flexibility is determined by the choice of the edit-time grammar. In the following sections, we describe how the relaxations in JPie's edit-time grammar enable mixed-mode editing.

4.2.1 Empty Expressions

As in graphical editing, empty expressions provide the programmer with the freedom to choose the order in which to fill in sub-expressions. This is convenient in graphical editing, but is absolutely essential in textual editing. A text editor that forced programmers to fill in every sub-expression in a prescribed order would be highly modal and very inflexible.

4.2.2 Type Mismatches

As in graphical editing, type mismatches in textual editing allow the incremental construction of expressions. As an example, consider the construction of a *chain expression*, which we define to be a string of variable and method accesses like

```
variable.method1().method2()
```

that might be found in a language like Java. During the construction of the chain expression, intermediate states like `variable.method1()` might not match the expected type. However, the model must be updated after these intermediate states are completed so that the environment can enforce syntactic safety. For instance, in our example the environment must know that the programmer is accessing a method on the return type of `method1()` to know whether

```
variable.method1().m
```

could lead to a syntactically correct expression. This required model update can occur only if type mismatches are allowed in the edit-time grammar.

4.2.3 Empty Operators

An empty operator is a placeholder for an undetermined infix operator expression whose operands may be specified. Empty operators are a relaxation made specifically for mixed-mode editing. Empty operators facilitate textual editing patterns that involve an intermediate state in which two operands of an infix operator expression are juxtaposed.

For example, one of the common editing patterns found in [11] is that of creating an infix operator expression when the two operands are already present, as in transforming `foobar` to `foo+bar`. This edit is an intermediate state in changing one infix operator to another and in prefixing an expression with an operand and an operator. Allowing an empty operator between `foo` and `bar` (so that the text is `foo bar`) should be allowed to support these edits.

4.3 Implementation

The relaxed edit-time grammar implemented in JPie supports the features outlined in this section. Consistency between text and graphics is maintained using a hierarchical "editor tree" that links the textual representation of expressions to JPie's internal model. Each editor responds to keyboard input differently, depending on the semantics of the underlying model.

5. Evaluation

JPie has been used for four semesters in an introductory computer science class for non-majors at Washington University. Students in the class use JPie as a tool to explore fundamental computer science concepts. Informal observations and student evaluations indicate that the curriculum [3] has been successful for a wide demographic of students.

Students take advantage of the flexibility of the edit-time grammar to modify code and to construct portions of a solution with placeholders. They have been able to create fairly involved projects (like a client-server chat program) in one or two ninety-minute classes. Their questions overwhelmingly focus on program logic or design instead of syntax, providing evidence that JPie is supportive enough to allow beginners to focus on higher-level ideas.

Students often fix discrepancies at runtime and continue execution. However, they sometimes restart programs after inadvertently terminating executing threads that "pop up" in the JPie debugger window because of a discrepancy. To encourage users to take full advantage of live execution, we are considering delaying the appearance of the debugger until the current sequence of edits is completed.

Members of JPie's development team have used mixed-mode editing to create programs during testing and course development. These more experienced users often prefer mixed-mode editing to graphical editing because of its increased flexibility. Because mixed-mode editing is a recent addition to JPie, it has not yet been evaluated in a classroom setting.

6. Future Work

We plan to conduct usability tests and a formal study of JPie as an educational tool when it is introduced in the introductory computer science class for majors at Washington University. These tests could be conducted in terms of the Cognitive Dimensions framework [7] and could be used to compare the usability of mixed-mode editing to that of textual programming environments like Eclipse.

7. Conclusion

Relaxing the edit-time grammar facilitates a balance between structure and flexibility in the editing process. Live development helps to minimize the impact of discrepancies between the edit-time and runtime grammars. As evidenced by mixed-mode editing in JPie, a relaxed edit-time grammar permits the integration of textual editing into a DMPE, both to enhance flexibility and to support students in their transition to textual programming environments.

8. Acknowledgements

We thank all past and present members of the JPie development team. We are grateful to Ben Brinckerhoff for his contributions during the design and implementation of mixed-mode editing. We also thank Ben Brinckerhoff, Sajeeva Pallemulle, and Joyce Santos for helpful comments on drafts of this paper. This work was supported in part by National Science Foundation grant 0305954.

9. References

[1] B.H. Brinckerhoff and K.J. Goldman, *Learning Curve Management in Educational Programming Environments*, tech. report TR-2004-78, Dept. of Computer Science and Engineering, Washington University, 2004.

[2] S. Cooper, W. Dann, and R. Pausch, "Alice: A 3-D Tool for Introductory Programming Concepts," *Proc. 5th CCSC Northeastern Conference*, Consortium for Computing in Colleges, 2000, pp. 107-116.

[3] K.J. Goldman, "A Concepts-First Introduction to Computer Science," *Proc. 35th Technical Symposium on Computer Science Education* (SICCSE 04), ACM Press, 2004, pp. 432-436.

[4] K.J. Goldman, "An Interactive Environment for Beginning Java Programmers," *Science of Computer Programming*, vol. 53, no. 1, October 2004, pp. 3-24.

[5] K.J. Goldman, *Capsules and Semantic Regions for Code Visualization and Direct Manipulation of Live Programs*, tech. report TR-2004-79, Dept. of Computer Science and Engineering, Washington University, 2004.

[6] K.J. Goldman, *Live Software Development with Dynamic Classes*, tech. report TR-2004-81, Dept. of Computer Science and Engineering, Washington University, 2004.

[7] T.R.G. Green and M. Petre, "Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework," *Journal of Visual Languages and Computing*, vol. 7, no. 2, 1996, pp. 131-174.

[8] C. Kelleher et al., "Alice2: Programming without Syntax Errors," *Proc. 15th ACM Symp. User Interface Software and Technology* (UIST 02), ACM Press, 2002.

[9] C. Kelleher and R. Pausch, *Lowering the Barriers to Programming: a survey of programming environments and languages for novice programmers*, tech. report CMU-CS-03-137, School of Comp. Science, Carnegie Mellon, 2003.

[10] A.J. Ko, "Designing a Flexible and Supportive Direct-Manipulation Programming Environment," *Proc. 2004 IEEE Symp. Visual Languages – Human Centric Computing* (VLHCC 04), IEEE Computer Society, 2004, pp. 277-278.

[11] A.J. Ko, H. Aung, and B.A. Myers, "Design Requirements for More Flexible Structured Editors from a Study of Programmers' Text Editing," *Extended Abstracts CHI 2005: Human Factors in Computing Systems*, ACM Press, 2005.

[12] M. Read and C. Marlin, "Generating Direct Manipulation Program Editors within the MultiView Programming Environment," *Joint Proc. 2nd Int'l Software Architecture Workshop and Int'l Workshop on Multiple Perspectives in Software Development*, ACM Press, 1996, pp. 232-236.

[13] B. Schneiderman, "Direct Manipulation: A Step Beyond Programming Languages," *IEEE Computer*, vol. 16, no. 8, August 1983, pp. 57-69.

[14] T. Teitelbaum and T. Reps, "The Cornell Program Synthesizer: a Syntax-Directed Programming Environment," *Communications of the ACM*, vol. 24, no. 9, September 1981, pp. 563-573