ProfessorJ : A Gradual Introduction to Java through Language Levels

Kathryn E. Gray Matthew Flatt
University of Utah

Abstract

In the second-semester programming course at the University of Utah, we have observed that our students suffer unnecessarily from a mismatch between the course content and the programming environment. The course is typical, in that it exposes students to Java a little at a time. The programming environments are also typical, in that they report compilation and run-time errors in the jargon of professional programmers who use the full Java language. As a result, students rely heavily on teaching assistants to interpret error messages, and valuable classroom time is wasted on syntactic diversions.

ProfessorJ is our new programming environment that remedies this problem. Like other pedagogical environments, such as BlueJ and DrJava, ProfessorJ presents the student with a simplified interface to the Java compiler and virtual machine. Unlike existing environments, ProfessorJ tailors the Java language and error messages to the students' needs. Since their needs evolve through the course, ProfessorJ offers several language levels, from Beginner Java to Full Java.

Categories and Subject Descriptors

K.3 [**Computer and Information Science Education**]: Computer science education

General Terms

Design, Human Factors, Languages

1 Introduction: Languages and Pedagogical Environments

A student in a first- or second-semester programming course faces two tasks: learning general programming principles (data structures, functions, objects, modularity, testing, etc.), and learning the notation of a particular programming language in which to express those principles. Instructors typically prefer to emphasize principles, since a student with a firm grasp of principles can ultimately adapt them to any programming language. Nevertheless, especially in the first few courses, learning a particular language is an important step for the student. The language's strict syntactic and semantic rules, faithfully implemented by the compiler and run-time system, reinforce the idea of computation as a well-defined process, and that computers really "do only what you tell them to do."

One problem for the student is that compilers and run-time systems *don't* faithfully implement the syntactic and semantic rules that are presented in class. Most programming environments implement a "large" language, such as Java, and no reasonable course begins by explaining all of the Java language. Instead, the instructor typically defines a subset of Java that students use for the first few days or weeks. This subset is gradually expanded throughout the semester, so that by the end of the course the language has been presented as a whole. In the meantime, however, students face a troublesome mismatch between the language presented in class and the language provided by the programming environment.

This mismatch is most troublesome with respect to error messages. For example, early in a Java-based programming course, a student might write something like the following:

```
int convert(int in, int by) {
...}
...
convert(4);
```

Most Java environments respond with an error message indicating that the method `convert` cannot be found (sometimes including the type of the given argument in the message). Students who have not yet learned about overloading are baffled by the compiler's claim that "convert" cannot be found. As another example, consider the error that results from a simple typo:

```
pulic class List { ...}
```

The error message "'class' or 'interface' expected" confuses students, because they see that `class` is in fact provided, and they do not see the misspelling of "public." Attempting to correct the problem by trial and error, a student will often simply delete "pulic", and then encounter further confusion later when the class is inaccessible in another package.

Students are needlessly confused by inappropriate error messages with all of the Java programming environments that we have tried

OOPSLA'03, October 26-30, 2003, Anaheim, California, USA.

in our course. The solution is to create a programming environment that is adapted to pedagogical needs. Although a few such environments exist for Java—notably BlueJ [10, 11] and DrJava [1]—these environments have not addressed the problem of taming the Java language to provide a subset for teaching. A programming environment that faithfully implements a series of pedagogical language subsets can reduce confusion for students, save instructor time in explaining not-yet-relevant language details, and generally encourage students to think in terms of well-defined behavior instead of black-box tinkering.

ProfessorJ, our new programming environment for Java, provides such a series of languages. It builds on the success of DrScheme [4], which implements a series of Scheme-like languages, and which we use in our first-semester course at Utah. We developed ProfessorJ for use in our second-semester course, which covers data structures, basic algorithms, and Java. ProfessorJ's language levels are designed specifically to avoid much of the confusion that we have observed in past offerings of the course. We will use ProfessorJ to begin teaching these concepts at the end of our first course for the 2003/2004 academic year.

In section 2, we describe the predecessor course to our Java course, and how it relates to ProfessorJ's design. In section 3, we describe ProfessorJ's language levels, and we explain how the student behavior we observed led to our choices. In section 4, we describe ProfessorJ's implementation. Section 6 describes related work.

2 Introductory Course Sequence

Most students entering our second-semester course have been introduced to programming in the first-semester course using *How to Design Programs* [3]. This curriculum covers program design in a largely functional style using a subset of Scheme. The specific language used in the first course is less important for our purposes than the course's data-centric approach to design and its emphasis on pedagogical environment support.

A "data-centric" approach means that we teach students to first understand the data representation for a problem, and then to allow the shape of the data to drive the rest of the design. A canonical example from early in the semester is the "list of numbers" datatype, which might be used to represent a list of prices in a toy store:

> A *list-of-numbers* is either
>
> - `empty`
> - `(cons` *number list-of-numbers*`)`

The *list-of-numbers* data definition drives the implementation of an `inventory-value` function that consumes an instance of the datatype. In particular, the function's implementation should match the shape of the data: two cases, handling a compound data value in the second case, and with a self-reference in the second element of the second case:

```
(define (inventory-value l)
  (cond
   ((empty?  l) ...)
   ((cons?  l) ...  (first l)
               ...  (inventory-value (rest l)))))
```

This data-oriented approach in the first-semester course transitions naturally to an "object-oriented" approach. A data definition with two cases corresponds to an abstract base class with two subclasses. The two `cond` lines in `inventory-value` turn into separate method implementations in the subclasses, with a method invocation in the second one.

```
abstract class Inventory { abstract int Value(); }

class Empty extends Inventory {
  Empty() { }
  int Value() { return 0; }
}

class Addition extends Inventory {
  int val; Inventory rest;
  Addition(int v, Inventory r) { val = v; rest = r; }
  int Value() { return val + rest.Value(); }
}
```

Our goal is to manage this transition as seamlessly as possible, keeping the focus on program design, rather than the arbitrary details of a particular compiler or programming environment.

To that end, ProfessorJ uses the same graphical user interface as DrScheme, as shown in Figure 1. The GUI presents a single window with two nested windows, called the *definitions window* and the *interactions window*. The definitions window (the upper one) contains a program that students can execute, save to disk, and access in the interactions window. The interactions window (the lower one) provides a "read-eval-print loop" (REPL), which students use to experiment with their programs and with the language constructs. Both windows implement the same programming language, use the same error messages, and report result values using the same syntax.

The ProfessorJ environment differs from DrScheme in one key way. In DrScheme, students can write the same code in either window, essentially because Scheme is an expression-oriented language. In ProfessorJ, the definitions window must contain only declarations, such as a class declaration, and the interactions window evaluates only statements and expressions (except for the `this` expression, which has no meaning outside of a class declaration). ProfessorJ allows multiple public class declarations in the definitions window. As in DrScheme (and DrJava [1]), students using ProfessorJ can access the classes written in the definitions window within the interactions window after pressing the Execute button. Each of the levels can interoperate with the others, which allows students to reuse implementations in earlier levels, and allows instructors to provide full support libraries implemented in full Java.

3 Language Levels

To accommodate the material presented in our second course, we have designed ProfessorJ with three language subsets: Beginner, Intermediate, and Advanced. Students will use Beginner Java for a few days as an introduction to Java. Students will then move to Intermediate Java as the course introduces new concepts, starting with the fundamentals of object-oriented programming (as a refinement of the first semester's data-oriented design) and object polymorphism. Eventually, the data structures and algorithms portion of the course requires that students move to Advanced, which introduces loops and arrays. Finally, the course ends with full Java (though with error messages that our students can understand) to present concepts such as exception handling.[1]

[1] The exact content of these language levels will likely evolve as experience in the classroom suggests changes.

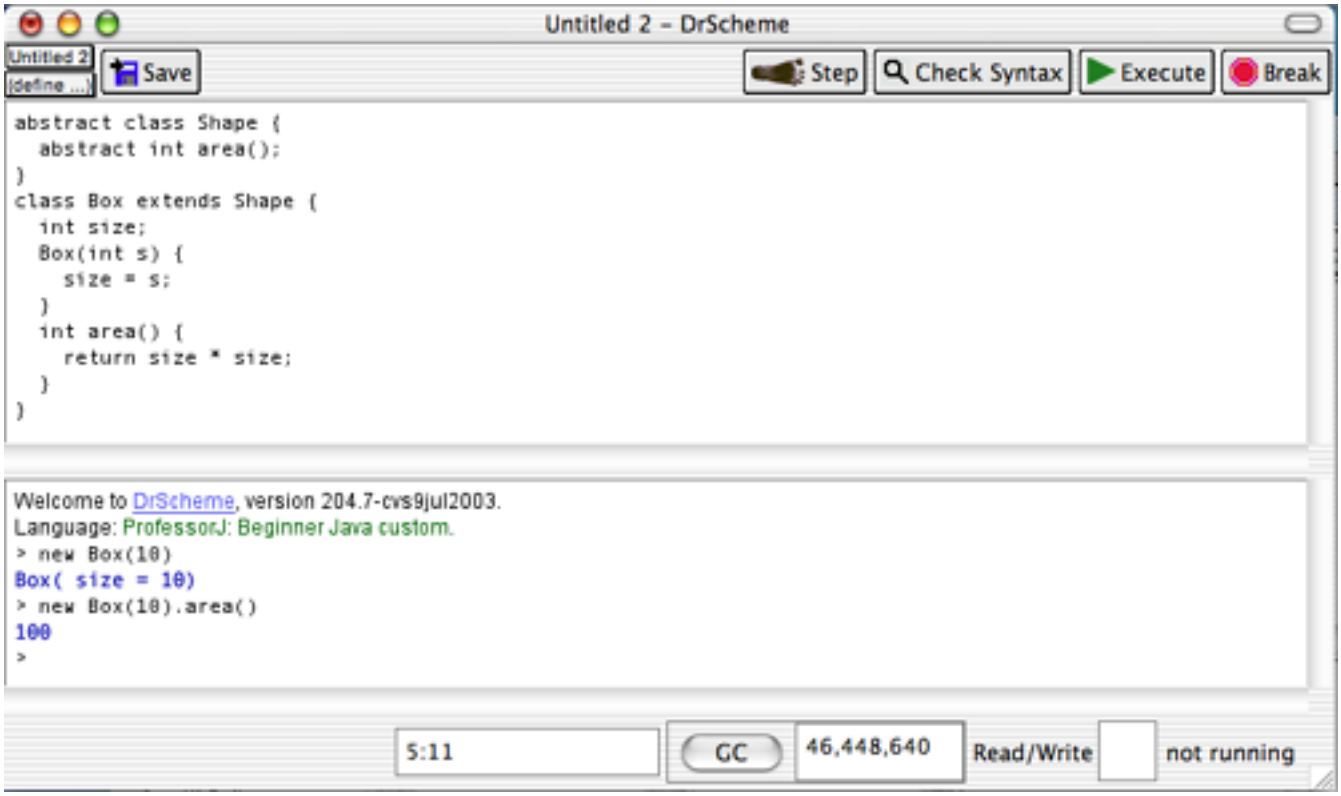

Figure 1. DrScheme with ProfessorJ

Construct	Restrictions
imports	Some imports not allowed
classes	Implicitly public Cannot be final Cannot have static members
fields	Implicitly private Implicitly final Must be set in constructor
constructors	Implicitly public Cannot be overloaded May contain only assignments
methods	Implicitly public Cannot be overloaded Cannot return void
Exclusions: package, interfaces, class and instance initialization blocks, inner classes and interfaces, all modifiers, arrays	

Figure 2. Beginner Java Declaration Constructs

3.1 Beginner

The Beginner level is our students' first exposure to Java syntax. As such, we selected the declaration constructs (see Figure 2) to provide enough structure to write programs and experience the general flavor of Java syntax. While not its primary purpose for now, we hope that Beginner will also be suitable for first-semester courses using Java.

The minimalist nature of Beginner allows students to familiarize themselves with core Java constructs while writing only familiar functional, recursive methods. The statements and expressions (see

Construct	Restrictions
Statements	
if	Must have else
return	Must have expression
assignment	Cannot be +=,-=, etc. Must be in constructor
Excluded statements: block of statements, variable declaration, throw, while, do, for, try, switch, break, continue, label, synchronized, ++ and –	
Expressions	
Literals	
this	
Binary operations	+ may not be used as string append
Unary operations	++, -- not allowed
Variable reference	
Field access	
method call	
class allocation	
Excluded expressions: cast, qualified name access, array access, array allocation, array instantiation, ? conditional, instanceof, assignment	

Figure 3. Beginner Java Statements and Expressions

Construct	Restrictions
imports	Some imports not allowed
classes	Implicitly public Cannot be final Cannot have static members
interfaces	Implicitly public Cannot have static members
fields	Implicitly private Cannot be final
constructors	Implicitly public Cannot be overloaded
methods	Implicitly public Cannot be overloaded
Exclusions: package, class and instance initialization blocks, inner classes and interfaces, field and method modifiers, arrays	

Figure 4. Intermediate Java Declaration Constructs

Figure 3) are all semantically familiar to students.

Static members are excluded so that a distinction between instance members and class members is not required at this stage of the course. Due to the REPL, static members are not necessary to run programs. Member modifiers are also excluded to simplify the language.

We expect students to use classes in implementing the same data structures that they have already implemented in Scheme. These Scheme structures are groupings of data that should map easily to objects with immutable fields, where the field values are given as constructor arguments. To facilitate this mapping, we require that fields, if present, be set in the constructor, and we do not allow further mutation to a field. Constructors do not have `super` calls because the data structures will not require inheritance of non-abstract classes. The Intermediate level will introduce the notion of calling the superclass constructor.

As evident in Figure 2, methods may not be overloaded or declared `void`. Overloading is a new concept to our students, and so it is not permitted in the languages until it is taught in the course. Methods may not return `void`, since we do not allow mutation in Beginner, and there would be no point in having a method which could return `void`. This reasoning extends to require that each `return` statement returns an expression.

For the remaining statements and expressions, Beginner imposes restrictions to avoid some of the more confusing errors that we have observed our students make. One such error occurs with overloaded `+` on strings. Since overloading has not been explained to students, this operation is potentially confusing, and no functionality is lost by removing it.

The `if` statement requires an `else` branch as many students forget to include the keyword `else`. In such cases, when the student's method executes, both the `then` and expected `else` behavior occurs. Often, only the effects of the `else` are visible. This error tends to cause the student to question the condition expression, since compilation indicates that the syntax is correct. In later language levels, this requirement may cause students to write dead code, but protection from odd behavior is worth a small amount of excess code.

Construct	Restrictions
Statements	
if	Must have else
return	
block of Statements	
assignment	Cannot be +=,-=, etc.
method call	
variable declaration	
Excluded statements: throw, while, do, for, try, switch, break, continue, label, synchronized, ++ and --	
Expressions	
Literals	
this	
Binary operations	+ may not be used as string append
Unary operations	++, -- not allowed
Variable reference	
Field access	
method call	
class allocation	
cast	
Excluded expressions: qualified name access, array access, array allocation, array instantiation, ? conditional, instanceof, assignment	

Figure 5. Intermediate Java Statements and Expressions

3.2 Intermediate

The Intermediate level presents a subset suitable for teaching students object-oriented design, complete with object polymorphism and dynamic dispatch. Additionally, this level provides fully extendible classes, interfaces, overrideable methods, and class instantiation (see Figure 4). With the exception of interfaces, these elements are necessary to present the desired concepts. We include interfaces to demonstrate a different axis of inheritance, and to ensure that students are exposed to this key Java construct.

Interfaces are potentially confusing, however, in our experience. We have seen cases where a student adds a method to a class that is not in the implemented interface. The student passes the object into a method with the interface type, and attempts to access their new method, since it certainly exists for the particular class. The typical error message confuses students as it merely indicates that the method is not found. This confusion is minimized in ProfessorJ by providing an error message that clearly states the known type of the object as the interface, and states that the interface does not contain the method.

As shown in Figure 4, classes and methods must be public while fields must be private. This restriction accomplishes two purposes. The first is to ensure that students program with the style of hiding fields and exposing appropriate methods. The second is to prevent confusing errors and potential mistakes by allowing students to create members with accessibilities that they do not fully understand.

If students can choose the accessibility of their method or field, then they might encounter behavior that is difficult to track down. Initially, students do not understand the distinction between protected and public, and might attempt to use a protected method from an unrelated class because the method was accessible before from a subclass. We have observed that this confusion leads to misunderstandings regarding why the method is not found. Such distinctions will be introduced later in the course, after students understand in-

heritance and method dispatch.

Like Beginner, Intermediate excludes static members. With the REPL, the only reason to have static members is to encode class-specific data and functionality. We have observed that students do not initially understand the distinction between class-specific and instance-specific needs, and they routinely declare fields or methods incorrectly. When this mistake occurs with fields, the wrong data is accessed, and this causes errors that can be difficult to debug. When methods are accidentally declared static, the student learns to pass information into the method (such as an object) to regain functionality, at the cost of readability and object-oriented style. We therefore choose to present static members later in the course, after more fundamental OO concepts.

3.2.1 Intermediate Restrictions

The remaining statement and expression restrictions (see Figure 5) help avoid mistakes that are common at this point in the course. Experience with other languages assisted in these decisions.

The particular requirement that the `if` statement contain an `else` statement continues from Beginner. While students in Intermediate are more familiar with Java syntax, our experience suggests that students are likely to continue to exclude the else keyword.

Assignment may not be used as an expression, which prevents the following code:

```
if (x = false) ...
```

The implications of this statement are obvious, and are acceptable within Java because experienced programmers would not test a boolean using `==`. However, beginning students often write `(x==true)` or `(x==false)`. One reason is that these expressions visually reinforce the desired value of x. Students should be encouraged to test their booleans using other expressions, but the language should not allow them to make this particular mistake.

The restrictions discussed so far control the behavior of constructs included in Intermediate. Many other Java constructs have been left out of the language, because we gradually expose students to new concepts in the course. Removing arrays, exceptions and overloading allows students to concentrate solely on object-oriented design at first.

Imports are restricted so that students cannot access classes that could encourage non-object-oriented programming (such as those classes implementing reflection), or that could cause errors they do not understand. Without restrictions, students could import and use classes that appear to perform operations they desire (such as graphical display, I/O, or reflective operations). But, exceptions and errors from these classes are not designed to accommodate their knowledge level. Further, import restrictions can prevent students from using libraries that have been disallowed for particular assignments. This restriction is useful when students are implementing data structures and algorithms for which Java libraries exist. However, we do not wish to forbid all imports, as they allow students to access support code or instructor-provided graphical programs.

3.2.2 Intermediate Omissions

Students are not allowed to use methods or classes that present reflection. This restriction ensures that students do not rely on reflexive properties to determine their types or attempt to perform operations in non object-oriented ways.

The `instanceof` expression is withheld for similar reasons. With access to `instanceof`, students write functional-style methods, instead of using method dispatch.

Construct	Restrictions
package	
imports	Some imports not allowed
classes	Cannot be final
interfaces	
arrays	
fields	Cannot be final
constructors	
methods	
class initializers	
Exclusions: class initialization blocks, inner classes and interfaces	

Figure 6. Advanced Java Declaration Constructs

3.3 Advanced

The Advanced level primarily introduces two new constructs: loops and arrays. The other added constructs (see Figure 6) prepare students to move to full Java. Many of the restrictions from Intermediate are removed to allow more flexible implementations, including most of the restrictions on statements and expressions (see Figure 7). Other restrictions remain to protect students from errors they are still likely to make.

As mentioned above, Intermediate does not allow static members or member modifiers as students are unsure how to use them. After working with Intermediate, students will understand the general nature of object-oriented programs. With this knowledge, students should be able to reason about the accessibility needs of a given member.

For advanced, we also removed the restriction against overloaded methods, so that students will be able to write and design more complicated software. However, overloaded methods are still error prone, as students attempt to call the method with slightly incorrect arguments. We attempt to solve this problem through reporting different error messages for different types of method-lookup failure. In other environments, when a programmer supplies the wrong number of arguments to an overloaded method, the error message reports that the method is not found. DrJava, for example, reports "No 'foo' method in 'Example"' for any method lookup error involving "foo", which is confusing to students who can see the method definition. ProfessorJ provides one of three messages. If foo was not called with any correct number of arguments, the message is "No definition of Foo with 1 argument(s) is found." A similar message indicates that the types of the arguments are incorrect. This should provide enough support for students to understand their errors with overloaded methods.

Advanced also introduces packages. Advanced students will write larger programs that can benefit from package-level grouping. This will also prepare them for designing Java software.

Exceptions, inner classes, and goto-like breaks and continues are the primary constructs withheld in Advanced. These elements are too complicated to be presented to students at this stage, and they are unnecessary for the programs that students write.

Construct	Restrictions
Statements	
if	
return	
block of Statements	
assignment	
method call	
variable declaration	
while	
for	
do	
break	inside a loop
continue	inside a loop
Unary operations ++ & --	
Excluded statements: throw, try, switch, label, synchronized	
Expressions	
Literals	
this	
Binary operations	
Unary operations	
Variable reference	
Field access	
Array access	
method call	
class allocation	
array allocation	
array initialization	Cannot be anonymous
cast	
instanceof	
? conditional	
Excluded expressions: qualified name access, assignment	

Figure 7. Advanced Java Statements and Expressions

Our experience suggests that unless class time is spent on the proper use of exceptions, students tend to over-use them. This time fits best into our schedule after the algorithmic material on loops and arrays for which Advanced is designed.

4 Implementation of ProfessorJ

To implement language levels, we had several options: modify an existing Java compiler, extend an existing Java environment, or extend DrScheme. We rejected the first option, as a full graphical environment offers many benefits to students. While the second option would allow users to import libraries that exist only in bytecode, and potentially would have allowed the reuse of existing Java compiler components, there were more advantages to extending DrScheme. Although building a Java environment on top of a Scheme environment may seem like a surprising choice, DrScheme is designed to accommodate non-Scheme languages as well as variants of Scheme.

Since our students are familiar with DrScheme, they do not have to learn how to navigate a new environment. Students will be able to use existing tools and features in DrScheme that interoperate with the ProfessorJ extension. Additionally, as the Java extension can interoperate with Scheme, Java programs can use Scheme programs, allowing students and teachers to use existing Scheme programs in their Java assignments.

The DrScheme tools and features that benefit ProfessorJ include advanced syntax highlighting, thorough error-highlighting, and the ability to arbitrarily break execution. To our knowledge, these features are not uniformly present in existing Java environments, and they are not easily implemented.

While many environments provide syntax highlighting for Java, they primarily highlight keywords and types. ProfessorJ, building on DrScheme's check-syntax tool, highlights program variables, and can draw arrows from the binding instance of a variable to all of its uses. We plan to extend this capability to connect uses of a class or interface type to its declaration. These arrows will allow students to track their variables and assists in their understanding of the behavior of their program.

Highlighting the source of an error simplifies the task of debugging, drawing students quickly to the most likely source of their problems. Most Java environments that we have used provide this functionality for compile-time errors. However, performing such service for runtime errors also benefits students, and we found that many existing environments do not support such highlighting outside of a debugging environment. Also, we wanted to support this highlighting in the interactions window as well as the definitions window, which other major pedagogical environments do not support.

DrScheme allows users to arbitrarily break evaluation at any time. All of the other Java programming environments that we have used only stop evaluation at set break-points in a debugging mode. Programs in all of the ProfessorJ languages can be arbitrarily stopped, due to the features of the underlying system. Reliable stopping has at least two benefits: if an infinite loop is encountered while some code has not been saved, the loop can be terminated without closing the environment and losing the work; when an infinite loop is encountered, it can be stopped right away to determine the code being run, without waiting for the JVM to run out of memory or for the loop to occur again in a debugging mode.

ProfessorJ is also able to make use of new tools developed for DrScheme. For example, a new interactive test development environment is nearing completion. With this, students will be able to write their test cases, and have the answers checked automatically. Each of the ProfessorJ languages work correctly with this tool.

DrScheme's lack of a conventional debugger is one potential disadvantage for ProfessorJ. However, the full features of a professional strength debugger (as is available with other environments) are not needed when debugging introductory programs, and professional features increase the difficulties students have in using the debugger. Often, despite the presence of a debugger, we have seen students use other means to find their errors rather than attempt to use and understand a debugger that is more complicated then they require.

ProfessorJ acts as a "plug-in" for DrScheme. Java source is compiled into MzScheme [5] syntax. We compile from source so that the language can be properly restricted, with appropriate error messages. The drawback — that we cannot use precompiled Java bytecode — can be remedied in the future if necessary.

Java classes are compiled into MzScheme classes, which are similar to Java's. MzScheme classes extend only one parent, implement interfaces, have public and private fields, and have inheritable public and private methods. Other Java features (statics, protected members, and packages) are implemented through a combination of Scheme functions, modules, and hidden names. We implement overloading by creating names for methods that are based on the original name combined with the types (generating names that are illegal for Java programmers to write).

5 Experience

ProfessorJ is a work in progress. A preliminary version, with Beginner and Intermediate levels only, is available with DrScheme version 205 (downloadable from www.drscheme.org).

While ProfessorJ has yet to be used in our course, we tested it in a one-week workshop for high school and college teachers. Half of the workshop participants had attended a similar workshop covering 5 weeks of the curriculum of our first semester course, and the other half teach introductory programming in college.

Due to the participants prior knowledge of Java,[2] this experience does not demonstrate how effective ProfessorJ is in teaching OO design and algorithms to a typical class of second semester students. However, the experience demonstrated that the first two language levels, with slight modifications from those presented here,[3] sufficiently supported the concepts taught (OO design and OO data structures).

Participants, especially those previously uncomfortable with Java, found the error messages informative, and they noted that the language restrictions helped prevent them from making logical blunders that are legal programs in full Java. Some participants expressed displeasure with the early levels exclusion of arrays and loops.

The next major focus of our work will be to use ProfessorJ in the classroom.

6 Related Work

Among the many existing Java development environments, practically all are used in courses that teach introductory Java. The creators of some environments, such as JCreator [2], jGrasp [9], and TextPad [7], offer workshops and advice on how to use their environment in a classroom, but the environments were not designed for students. Only two environments other than ProfessorJ were designed for teaching: BlueJ and DrJava.

BlueJ [11], a commonly used Java pedagogic environment, focuses on teaching introductory students object-oriented programming while shielding students from some language complexities. To focus on OO programming, BlueJ graphically presents a collection of classes, with arrows denoting the inheritance and use relationships between them. Students can create classes by adding boxes in the environment, and they can build inheritance hierarchies by inserting arrows. Creating a class in this manner generates a bare bones class in the source code, with the class name and inheritance specified, as well as an example method and field. The method demonstrates the syntax for a method declaration (return type, name, and modifiers) and a return statement.

Once a class has been implemented and compiled, students can interactively create new instances, dispatch public methods and inspect fields. This functionality facilitates interactive testing (without writing specific test code or requiring an understanding of static methods), as well as visually presenting the inheritance of methods and fields. Newer versions of BlueJ also contain a statement and expression evaluator, where students can test code. BlueJ provides access to a graphical debugger, which sets breakpoints and allows users to step through execution (ala gdb).

Used only as an editor and interactive environment, BlueJ partially succeeds in shielding students from static members. Students do not have to write statics for their programs to work. However, because BlueJ does not have language levels enforcing this protection, students may still use static members (often incorrectly). Additionally, if students use the debugger, they are exposed to the concept of statics (and threads) before the material is presented in their course. A further problem with BlueJ, as has been discussed above, is that the error messages provided to students do not target their knowledge level. This problem has been mentioned in a user study [6] as one of the largest problems with BlueJ.

The graphical view provided by BlueJ is beneficial to students as they learn to design and reason about object-oriented programming. We are considering similar functionality for ProfessorJ.

DrJava [1] provides students with an environment where they can interactively experiment with Java, without needing to understand how to use an external compiler. As with DrScheme and ProfessorJ, DrJava has a definitions window and interactions window. Programs written in the definitions window are accessible to students in the interactions window. DrJava is also connected to a testing facility to assist students in debugging their code.

With respect to error messages in the definitions window, DrJava provides similar error messages as other Java environments. These

[2] Approximately 15% of the participants did not know any Java, approximately 30% had seen some Java but felt in-equiped to teach it, and 55% knew and felt prepared to teach it.

[3] To conform to the curriculum presented, fields default to public instead of private, and in Beginner all member data must be accessed using the keyword this.

messages are accurate and helpful for an experienced programmer. For a student, these messages do not use terminology the student understands, and the messages create the problems we have discussed previously. Furthermore, the error mechanism used (at this time) for the interactions window is not the same as for the definitions window, using even more difficult to understand messages.

The Expresso [8] project provides student-oriented error messages with their compiler, although Expresso is not a complete environment. The messages attempt to fully explain the error in terms students know. However, as this work does not introduce language levels, it is still possible for students to encounter errors about constructs they do not understand, and to write incorrect and difficult to debug programs.

7 Conclusions

ProfessorJ's provides the Java language to our students as they see it presented in the classroom, instead of Java as professional programmers must see it. Using ProfessorJ, we expect to spend more of our students' time on understanding concepts, and we expect to spend less time deciphering error messages and odd program behavior.

We will field-test ProfessorJ's Beginner and Intermediate levels this semester in the introductory programming course at Utah to help students transition into the second programming course. The Advanced and Full levels are nearing completion, and we expect to use these levels of ProfessorJ in future offerings of the second course.

8 References

[1] E. Allen, R. Cartwright, and B. Stoler. DrJava: A lightweight pedagogic environment for Java. In *SIGCSE Technical Symposium on Computer Science Education*, Sept. 2001. `www.drjava.org`.

[2] W. D. de Witte. *JCreator*. `www.jcreator.com`.

[3] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. *How to Design Programs*. The MIT Press, Cambridge, Massachusetts, 2001. `http://www.htdp.org/`.

[4] R. B. Findler, C. Flanagan, M. Flatt, S. Krishnamurthi, and M. Felleisen. DrScheme: A pedagogic programming environment for Scheme. In *Proc. International Symposium on Programming Languages: Implementations, Logics, and Programs*, pages 369–388, Sept. 1997.

[5] M. Flatt. PLT MzScheme: Language manual. Technical Report TR97-280, Rice University, 1997.

[6] D. Hagan and S. Markham. Teaching Java with the BlueJ environment. In *Ascilite*, 2000.

[7] Helios Software Solutions. *TextPad*. `www.textpad.com`.

[8] M. Hristova, A. Misra, M. Rutter, and R. Mercuri. Identifying and correcting Java programming errors for introductory computer science students. In *SIGCSE Technical Symposium on Computer Science Education*, Feb. 2003.

[9] J. H. C. II. *jGrasp*. `www.eng.auburn.edu/grasp/`.

[10] M. Kolling, B. Quig, A. Patterson, and J. Rosenberg. The BlueJ system and its pedagogy. In *Workshop on Pedagogies and Tools for Assimilating Object Oriented Concepts*, Oct. 2001.

[11] M. Kolling and J. Rosenberg. *BlueJ*. `www.bluej.org`.