Taming Java for the Classroom[*]

James I. Hsia — jhsia@alumni.rice.edu
Elspeth Simpson — elspeth@rice.edu
Daniel Smith — dlsmith@rice.edu
Robert Cartwright — cork@rice.edu

Rice University
6100 S. Main St.
Houston TX 77005

ABSTRACT

Java is the canonical language for teaching introductory programming, but its complex syntax and abundance of constructs are difficult for beginners to learn. This paper shows how object-oriented programming in Java can be made more accessible to beginners through the use of "language levels", a hierarchy of progressively richer subsets of Java. This hierarchy is implemented as an extension of the DrJava pedagogic programming environment.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environments

Keywords

DrJava, language levels, object-oriented programming

1. INTRODUCTION

Programming technology is in the midst of a paradigm shift. Object-oriented programming (OOP) in safe, *object-oriented* (OO) languages like Java and `C#` is gradually supplanting *object-based* programming in `C++` for mainstream applications. Many colleges have recently revised their introductory programming sequence (CS1/CS2) to use Java instead of Pascal or `C++`. High school courses in AP Computer Science made the transition from `C++` to Java last year. In the wake of this paradigm shift, computing educators face a dilemma about what concepts to teach in introductory courses using Java. Should they continue to teach the object-based perspective that has dominated programming curricula for the past decade? Or should they embrace a truly OO approach to program design for which there is little precedent? There is a growing consensus that OOP concepts should be taught in Java as early as possible [9].

Despite the growing consensus in favor of an "objects-first" curriculum, the implementation of this pedagogy has proven to be a challenge. The ACM Education Board recently formed a Java Task Force "to develop a stable collection of pedagogical resources that will make it easier to teach Java to first-year computing students without having those students *overwhelmed by its complexity* [emphasis added]" [1]. Java's complex syntax and plethora of language features are a serious obstacle to learning OOP and interfere with an early focus on OO concepts. Even the traditional first program, "Hello World", seems daunting to a beginner. To run "Hello World" from the command line, the student must write a main method whose declaration looks like this:

```
    public static void main(String[] args)
```

In this one declaration, the student is confronted with visibility modifiers, static methods, return types, and arrays—none of which are central to OOP. The instructor faces a choice between taking the time to explain what these complex, unrelated pieces of the language mean or telling the students to blindly cut and paste the code. Neither of these choices is very attractive.

To make Java more accessible to beginners, educators have developed pedagogic programming environments such as BlueJ [8] and DrJava [2] that support simpler, more intuitive programming interfaces. BlueJ provides a graphically based workbench where students can create objects and apply methods to them, while DrJava provides users with an Interactions Pane that transforms Java from a batch-oriented language to an interactive one with a read-eval-print-loop. Both of these interfaces completely eliminate the need to define a `main` method.

While these pedagogic environments are a major step forward over conventional Java environments, they do not fully shield students from the complexities of Java syntax or necessarily encourage OO design. Nearly all Java texts, including those that presume the support of a pedagogic environment like BlueJ or DrJava, cover the major syntactic elements of procedural programming—namely assignment, conditionals, and looping constructs—plus Java language features like `static`, `final` and visibility modifiers, before discussing the essence of OOP: polymorphism (dynamic dispatch). The detour through the mechanics of procedural programming is time-consuming and distracting, making it difficult to teach an OO perspective on program design.

What is needed is a hierarchy of Java *language levels*—progressively richer subsets of the Java language that facilitate a focus on OO design rather than the mechanics of the Java language. The idea of partitioning a language into a hierarchy of language levels dates back to at least the 1970's when Richard Holt *et al* developed the SP/k framework for teaching structured programming in PL/I [7] at the University of Toronto. More recently, the DrScheme environment developed by Matthias Felleisen and his students at Rice

[*]This research has been partially supported by the Texas Advanced Technology Program and the National Science Foundation.

University pioneered the idea of partitioning a language into a *semantic* hierarchy of language levels. In DrScheme, the language level hierarchy supports a corresponding hierarchy of computational models, each with a richer data model and more complex semantics than its predecessor. As a result, the progression of language levels in DrScheme directly corresponds to a progression of programming abstractions suitable for teaching. The pedagogy underlying this progression is explained in the textbook *How to Design Programs* (HTDP)[4].

Although HTDP focuses on “mostly” functional programming in Scheme, the underlying pedagogy is largely language independent. In fact, the same programming concepts and principles can be articulated in the context of OOP in Java. In HTDP, program design is *data-directed*, primarily using inductively-defined *algebraic* types like lists and trees.[1] Given the inductive definition for a data domain, there is a corresponding, mechanically generated template for writing a function to process that form of data. In addition, the program development process is *test-driven*; test cases are written for each program function before it is coded.

In an OO context, the design concepts in HTDP correspond to progressively more sophisticated uses of polymorphism that have been codified as *design patterns* in the well-known “Gang of Four” tome [5] on the subject. It is remarkable that two largely disjoint programming cultures have distilled essentially the same principles of program design. The programming methodology from HTDP looks equally familiar when it is translated to an OO context: it is simply Extreme Programming (XP) [10] in-the-small: test-driven development using JUnit.

In this paper, we describe a language levels framework that we have developed for the DrJava programming environment. This framework supports our translation of the programming pedagogy from HTDP into an OO context. With this framework, OO programming in Java is easily accessible to beginners, as demonstrated by the simplicity of the OO program written at the Elementary level in Figure 1.

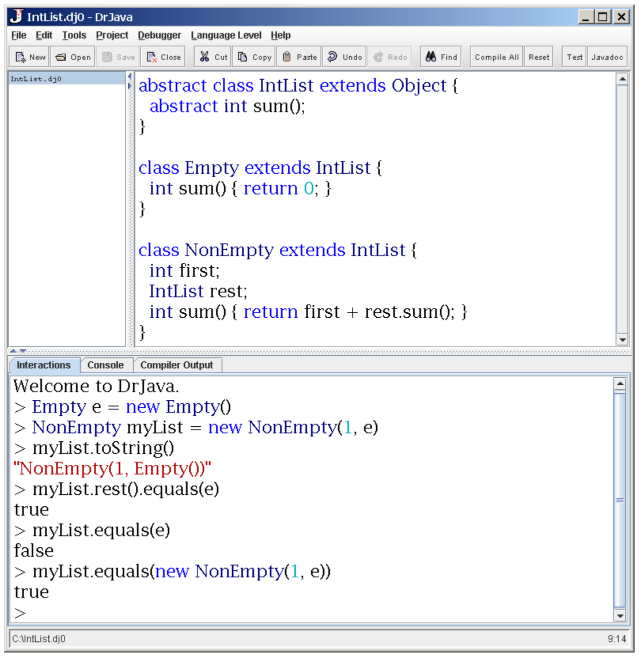
2. AN OVERVIEW OF DRJAVA

DrJava [2] is a free, open-source, lightweight IDE developed at Rice University by undergraduate and graduate students under the direction of Prof. Robert Cartwright. The user interface is designed to be simple and highly interactive. There are only three panes: a Navigator Pane lists all open documents, a Definitions Pane displays the selected document (typically a file defining a Java class), and an Interactions Pane allows the user to evaluate arbitrary Java program text. The Definitions Pane supports the “intelligent” editing of Java program text through syntax highlighting, brace matching, and uniform indenting to help beginners adjust to Java syntax. The Interactions Pane encourages students to experiment with both their code and Java libraries by accepting an arbitrary series of program statements and expressions and evaluating them cumulatively “on-the-fly”. The value of each top-level expression is printed using the `toString()` representation of that value.

A small collection of menus on the title bar provides commands for basic actions, including:

- creating, opening, saving, and closing documents;

[1] A data type is *algebraic* if it can be defined by a *tree grammar* [3]. In OOP, algebraic types are defined by the *composite* pattern [5].


IntList.dj0 - DrJava
File Edit Tools Project Debugger Language Level Help
New Open Save Close Cut Copy Paste Undo Redo Find Compile All Reset Test Javadoc
IntList.dj0
Interactions Console Compiler Output
C:\IntList.dj0
9:14

```
abstract class IntList extends Object {
  abstract int sum();
}

class Empty extends IntList {
  int sum() { return 0; }
}

class NonEmpty extends IntList {
  int first;
  IntList rest;
  int sum() { return first + rest.sum(); }
}
```

```
Welcome to DrJava.
> Empty e = new Empty()
> NonEmpty myList = new NonEmpty(1, e)
> myList.toString()
"NonEmpty(1, Empty())"
> myList.rest().equals(e)
true
> myList.equals(e)
false
> myList.equals(new NonEmpty(1, e))
true
>
```

Figure 1: DrJava at Elementary Level

- compiling and unit testing open document(s);
- enabling a source level debugger integrated with the Interactions Pane;
- running `javadoc` on all of the source files to generate HTML output;
- changing the configuration options for DrJava; and
- selecting a language level.

The most common commands are bound to buttons on a tool bar just below the title bar.

Although DrJava is targeted primarily at supporting beginning and intermediate programming courses taught in Java, it is also particularly well-suited to small production programming projects that use test-driven development. In fact, for the past two years, DrJava has been developed using DrJava. The combination of a general read-eval-print-loop (the Interactions pane) with tightly integrated support for unit testing and source level debugging provides an unusually responsive and productive environment for software development. To our knowledge, DrJava is the only IDE, commercial or open-source, that supports a full read-eval-print loop capable of evaluating arbitrary program text in the context of a debugger breakpoint environment.

The most recent release of DrJava fully supports Java 1.5 including generic types and the “Tiger” (JSR-201) language extensions (*autoboxing* and *auto-unboxing*, *foreach*, *enum* types, and *varargs*). It also includes a lightweight project facility capable of building projects like DrJava itself. CVS (the most widely used open source version control package) support has not yet been integrated in DrJava, but will soon be forthcoming. The primary features offered by professional IDE’s (like Eclipse and JBuilder) that are missing in DrJava are code completion and refactoring transformations.

3. PEDAGOGY AND LANGUAGE LEVELS

Before describing the design of our language levels framework for DrJava, we must explain an important aspect of our pedagogy that has shaped its design.

In our introductory programming curriculum, we follow the progression of programming concepts in HTDP, adapted to the context of OOP. As a result, we initially focus on programming with *immutable* data objects—objects where all fields are `final`. We impose this restriction for two reasons. First, programming with immutable data is easy for beginners because it is a natural extension of concepts from arithmetic and algebra learned in grammar school. Familiar laws such as the "substitution of equals for equals" hold in the context of immutable data, but break when mutation is added. Second, many computations depend on maintaining the immutability of data as an invariant. Mutating a data object is dangerous because it changes the state of every object that refers directly or indirectly to the mutated object. The integrity of some commonly used classes in the standard Java libraries depends on using immutable data. For example, only immutable objects can safely be used as keys in the `HashMap` and `HashTable` classes in `java.util`. The `String` class in `java.lang` is immutable for this reason.

In our experience as software developers, programming with immutable data confers so many advantages that we teach our students to scrupulously avoid mutation unless there is a specific justification for doing so. Two good reasons for using mutation are:

- **Faithful modeling**. When the entity modeled by a data object can change state, its data representation should be *mutable*. For example, the document objects that are edited in DrJava are mutable.
- **Application efficiency**. If data values are immutable, new values can only be created by explicitly constructing them. In contrast, mutation allows existing values to be updated to form new values. In some applications, the efficiency of the application critically depends on modifying existing values rather than allocating new ones. Consider a compiler that makes multiple passes over the AST representation for a source program. Each pass adds new attributes to the nodes of the AST. Building a new immutable AST on each pass is conceptually elegant, but is often not worth the overhead incurred.

In our framework, the first two language language levels enforce the immutability of data. This restriction guarantees that data structures do not contain cycles, enabling the language levels framework to automatically generate descriptive `toString()` methods for program classes.

4. LANGUAGE LEVELS DESIGN

Designing a language levels framework for Java is a more challenging problem than simply identifying an appropriate hierarchy of language subsets. Full Java requires programmers to write boilerplate methods such as constructors, selectors, and the `equals(...)` and `toString()` methods for simple algebraic data types like the `Empty` and `NonEmpty` list classes in Figure 1. To make programming with algebraic types accessible to beginners, two of our language levels automatically generate these boilerplate methods. Note that immutability plays a critical role in making this process tractable.

Our framework consists of three levels: Elementary, Intermediate, and Advanced, plus the full Java language (which DrJava already supports). Each successive level embodies a richer, more complex collection of abstractions for defining data and performing computation over that data. Nearly all of these abstractions are embodied as design patterns.

The following subsections briefly describe each language level and the associated programming concepts.

4.1 Elementary Level

The Elementary level focuses on computation over immutable algebraic data types such as booleans, integers, lists, and trees. As a result, executing programs is analogous to performing algebraic simplification. In an OO context, the natural representation of algebraic data follows the *Composite* design pattern where each clause of the inductive definition is represented as a concrete class extending an abstract class at the root of the composite hierarchy. Several other basic design patterns can also be taught at this level, including the *Union*[2], *Interpreter*, and *Factory Method* patterns.

Even though many important design concepts can be taught here, only a small subset of the Java language is allowed. Most importantly, data mutation is prohibited, discouraging the use of flags instead of dynamic dispatch. Loops and arrays are also prohibited since their normal usage requires mutation.

To reduce the number of keywords beginners must learn, the only keywords allowed other than the essential four (`class`, `if`, `else`, and `return`) are `this`, `abstract`, and `extends`. The explicit use of `this` naturally arises in some of our early uses of polymorphism, so we include it here. Since `abstract` methods, `abstract` classes, and the `extends` keyword are essential for polymorphism, the `abstract` and `extends` keywords must be supported. However, since beginning students often have difficulty understanding the distinctions between interfaces and abstract classes, we ban interfaces and the `implements` keyword at this level.[3] Many constructs that are not relevant to basic OOP are also prohibited, as shown in the table in Figure 3. Since `import` statements and fully-qualified class names are excluded, no libraries are accessible other than `java.lang` or those placed in the default package.

Significant code augmentation is done at the Elementary level. All fields and variables are made `private` and `final` to enforce the immutability of data, and all methods and classes are made `public`. For the sake of JUnit testing, if a class extends "TestCase", the necessary JUnit framework is automatically imported. Finally, all of the boilerplate methods supporting an algebraic view of data are automatically generated. This code includes accessor methods for all fields; method overridings for `toString()`, `equals(...)`, and `hashCode()` that produce results consistent with an algebraic view of data;[4] and a default constructor that takes in a value for each field of the class. This code augmentation significantly reduces the clerical burden on beginning students. Note the difference between code written at the Elementary level and the corresponding augmented code in Figure 2.

[2]We use the term *Union* to refer to a degenerate version of the *Composite* pattern in which there is no recursion in the definition.

[3]We arguably could use interfaces *instead of* abstract classes, but this precludes teaching method hoisting at this level since interfaces cannot include concrete methods.

[4]The `toString()` method returns a string giving the class name of `this` followed by the `toString()` representations of the fields enclosed in parentheses and separated by commas. The `equals(...)` method returns `true` if the argument is an instance of the same class as `this` and all respective fields are `equals(...)`; it returns `false` otherwise (assuming termination). The `hashCode()` method is overridden to be

```
Elementary Code:
   class NonEmpty extends IntList {
       int first;
       IntList rest;
       int sum() {return first + rest.sum();}
   }

Resulting Augmentation:
   class NonEmpty extends IntList {
       private final int first;
       private final IntList rest;
       public int sum() {return first + rest.sum();}

       public NonEmpty(int first, IntList rest) {
           this.first = first;  this.rest = rest;
       }
       public int first() {return first;}
       public IntList rest() {return rest;}
       public String toString() {
           return "NonEmpty("+ first + ", " + rest + ")";}
       public boolean equals(Object o) {
           if ((o==null) || getClass() != o.getClass())
               return false;
           NonEmpty cast = (NonEmpty) o;
           return first == cast.first && rest.equals(cast.rest);
       }
       public int hashCode() {return first ^ rest.hashCode();}
   }
```

Figure 2: Augmentation at Elementary Level

4.2 Intermediate Level

The primary language additions at the Intermediate Level are interfaces, `static` fields, visibility modifiers, `package` declarations, exceptions including `try-catch`, `throw` statements and `throws` clauses, anonymous classes, and explicit casts. Of these features, anonymous classes are the most important because they enable methods to pass "functions" (behavior) as data. In Java, anonymous classes play the same role as `lambda` expressions in functional languages.

The Intermediate level is designed to support the *Command*, *Strategy*, *Visitor*, and *Singleton* patterns, as well as the use of exceptions to signal program errors. Since multiple interface inheritance simplifies the coding of many common uses of some of these patterns, we include interfaces at this level. Since casts are often required to narrow the the output types of Visitor and Strategy classes, explicit casts are also allowed. We permit `static` fields because they are required for the Singleton pattern.

The inclusion of `package` and `import` statements gives students access to all the Java libraries, so the `null` constant (which can be returned by library methods) is allowed. Because students can place classes in named packages, this level requires explicit visibility modifiers for methods and classes so students can learn to distinguish between `private`, `package` and `public` constructs.

Since all program data is still immutable at this level, code augmentation is identical to that done at the Elementary Level, with three exceptions: static fields are automatically made public and final; default visibility modifiers are not generated for methods and classes; and the user must explicitly import `junit.framework.TestCase` to reference the JUnit `TestCase` class.

4.3 Advanced Level

Data mutation (re-assignment) is finally introduced at the Advanced Level. At this level, we teach students how to use procedural programming constructs such as loops, switch statements, and arrays. In addition, named inner classes and interfaces are allowed at this level.

consistent with `equals(...)`. See Figure 2 for an example.

The Advanced Level consists of full Java with the exception of synchronization, bitwise operators, and a few other constructs. Nearly all design patterns can be taught here, including the *State*, *Decorator*, and *Model View Controller* patterns. No code augmentation is done at this level.

4.4 Summary

Figures 3 and 4 summarize the language features and code augmentation for each level.

Language Construct	L1	L2	L3	Full
Classes	X	X	X	X
Non-void methods	X	X	X	X
fields, local variables	X	X	X	X
`abstract` modifier	X	X	X	X
`int`, `double`, `char`, `boolean` types	X	X	X	X
Core operators	X	X	X	X
`if` statement	X	X	X	X
Explicit constructors		X	X	X
`package` and `import` statements		X	X	X
`static` fields		X	X	X
Anonymous inner classes		X	X	X
Casts		X	X	X
Visibility modifiers for classes and methods		X	X	X
`null` value		X	X	X
Exceptions and try-catch statements		X	X	X
Interfaces		X	X	X
Visibility modifiers for fields			X	X
Assignment to fields, variables			X	X
Explicit use of `final` modifier			X	X
Nested classes, interfaces			X	X
`while`, `for`, and `do` loops			X	X
`switch` statement			X	X
`void` methods			X	X
Arrays			X	X
`break` and `continue` statements			X	X
instanceof operator			X	X
All other primitive types				X
Initialization blocks				X
`native` methods				X
`synchronized`, `volatile`, `Thread` classes				X
Bitwise operators				X
Labeled statements				X
Conditional operator				X

Figure 3: Features allowed at each level

Augmentation	L1	L2	L3
Methods, classes automatically `public`	X		
Static fields automatically `public`		X	
Instance fields automatically `private`	X	X	
Fields, variables automatically `final`	X	X	
Constructor generation	X	X	
`toString()`, `equals(...)`, `hashCode()`	X	X	
Accessors	X	X	
Concrete methods automatically `final`	X	X	

Figure 4: Code augmentation at each level

5. IMPLEMENTATION

DrJava translates language level source files based on the file extensions attached to file names. The Elementary, Intermediate, and Advanced levels are identified by the extensions `.dj0`, `.dj1`, and `.dj2`, respectively. This convention enables students to continue using code written at earlier language levels.

Implementation Architecture The translator for each language level maps a language level file to a compilable `.java` file in four stages. If the translator discovers serious errors at any point in this process, it aborts execution of the remaining stages and generates error diagnostics which are displayed for the user. We have placed an emphasis on generating understandable and relevant error messages.

In the first stage, which is identical for all language levels, the translator parses the student's source file into an *Abstract Syntax Tree* (AST), containing nodes for the core language constructs including class definitions, method invocations, variable declarations, and expressions. This parsing pass performs a coarse syntax check, looking for fundamental syntactic errors common to all language levels such as mismatched braces. If it encounters any fundamental errors, it reports them an aborts further translation. However, if the translator discovers an error embedded within an expression, (such as an omitted operator or operand), it encodes the error in the AST rather than immediately reporting it so that a *language-level-specific error message* can be given in the next pass. If no fundamental errors are found, the translator proceeds to the remaining three stages of the translation, which are performed by distinct visitors over the AST. Each "visitor" is implemented using the OO *Visitor* design pattern.

The second stage performs a language-level-specific `syntax check` and constructs a symbol table with information for all classes that are referenced directly or indirectly by the AST. This stage is implemented by a visitor that walks the AST, looking for illegal AST nodes (for example, the `public` keyword at the Elementary level or an inner class at the Intermediate level) and building a symbol table of referenced classes. If any errors are found, the visitor continues its pass over the AST to produce as many user diagnostics as possible. When it has finished, it gives a clear diagnostic for each error (if any) that was discovered.

The third stage performs type-checking. The type-checking visitor insures that all expressions are typed correctly and also performs some syntax checks that cannot be done until program expressions are typed. If any errors are found, the visitor finishes traversing the AST before displaying all error messages to the user.

The final stage performs code augmentation. Source code is copied in pieces from the student's language level file to a new `.java` file of the same name. The augmentation visitor traverses the AST as it copies source code, using the AST to determine where code augmentation is needed. The augmentations include the added modifiers, fields, and methods discussed in Section 4. The resulting `.java` file can now be passed directly to the `javac` compiler where it will compile successfully. More advanced students can also view the `.java` file to see their code's spacing and comments perfectly preserved, supplemented by the necessary augmentation.

Ensuring Reliability Writing and maintaining reliable software is difficult, because bugs can creep into programs of any size. Bugs are particularly pernicious in environments designed for beginners for two reasons. First, beginners have difficulty distinguishing aberrant behavior by the environment from mistakes in their own code or misuse of the environment. Thus, bugs in the environment may lead beginners to believe that they are incapable of writing programs. Second, beginners are much more likely to use unorthodox command sequences and strange coding conventions, stressing an environment in ways that the developers failed to anticipate and exposing hidden bugs.

Our strategy for achieving a high level of program reliability is to follow the tenets of Extreme Programming (XP) [10] with an emphasis on rigorous unit testing. Students and faculty in our introductory programming courses have admirably served as our *on-site customers*.

6. RELATED WORK

We know of only one other effort to support language levels for Java. Kathy Gray and Matthew Flatt at the University of Utah have developed ProfessorJ [6], a plugin for the DrScheme programming environment supporting a hierarchy of three Java language levels. ProfessorJ supports a more conventional Java pedagogy than we do. Mutation is allowed at all levels and no code augmentation is performed. Anonymous and local classes are not supported at any language level which makes it impossible to teach the *Visitor*, *Command*, and *Strategy* patterns in much generality.[5]

From the perspective of most educators using Java, the biggest problem with ProfessorJ is that it is implemented on top of DrScheme rather than a Java Virtual Machine. As a result, it is incompatible with Java binaries (class files) and does not support any of the standard Java libraries (including most of `java.lang`). For example, even the standard wrapper classes `Integer`, `Boolean`, *etc.* are not supported.

7. DIRECTIONS FOR FUTURE WORK

Our immediate goal is to determine which language extensions from Java 1.5 beyond auto-boxing/unboxing (which are already supported at all language levels) should be supported in the language levels framework. Later, we plan to develop a configuration facility for language levels, which will empower instructors to design language levels based on their own pedagogy rather than ours. One of the main issues that we will have to confront is whether we can support code augmentation in the presence of mutation because it breaks invariants on which our current code augmentation depends. At a minimum, we should be able to support configuration options that do not affect the status of mutation.

8. REFERENCES

[1] *ACM Java Task Force web site*, September 2004. `http://www.sigcse.org/topics/javataskforce`

[2] E. Allen, R. Cartwright, B. Stoler. *DrJava: A Lightweight Pedagogic Environment for Java. SIGCSE 2002*, March 2002. `http://drjava.org`

[3] H. Comon *et al. Tree Automata Techniques and Applications.* `http://www.grappa.univ-lille3.fr/tata`

[4] M. Felleisen, R.B. Findler, M. Flatt, S. Krishnamurthi. *How to Design Programs.* MIT Press, 2001.

[5] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[6] K. Gray, M. Flatt. *ProfessorJ; A Gradual Introduction to Java Through Language Levels. OOPSLA Educators Symposium 2003*, October 2003.

[7] R. Holt *et al. SP/k: a system for teaching computer programming. CACM* **20(5)**, 1977.

[8] M. Kölling *et al. The BlueJ system and its pedagogy*, *Journal of Computer Science Education* **13(4)**, 2003.

[9] B. Meyer. *Teaching Object Technology. TOOLS 11*, 1993.

[10] *XProgramming.com web site.* `http://xprogramming.com`

[5]Because all three patterns require forming *closures* in many cases, which are only supported in Java by local and anonymous inner classes.