AGUIA/J: A Tool for Interactive Experimentation of Objects

André L. Santos*†
andre.santos@iscte.pt

*Faculty of Sciences, University of Lisbon
LASIGE, Bloco C6, Piso 3, Campo Grande
1749-016 Lisboa, Portugal

†Instituto Universitário de Lisboa (ISCTE-IUL)
Av. das Forças Armadas, Edifício II
1649-026 Lisboa, Portugal

ABSTRACT

Learning and teaching object-oriented programming are still perceived as being difficult tasks. This paper presents AGUIA/J, a pedagogical tool for interactive experimentation and visualization of object-oriented Java programs. The approach is based on having a graphical environment for experimenting a set of user-developed classes where objects of such classes can be created and controlled interactively. The main innovative aspects of the tool comprise the visualization of objects in widgets that take different forms according to their classes and state, a mechanism to address the query-command separation principle, and the capability of runtime adaptation of the objects in the workbench to new versions of their classes. An experiment using AGUIA/J as courseware in pilot lab classes has resulted in higher approval rates for the involved students, as well as significantly lower drop-out rates.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education; D.1.5 [**Programming Techniques**]: Object-oriented Programming

General Terms

Design, Human Factors

1. INTRODUCTION

Programming is a complex activity that involves mental models that apprentices struggle to develop during introductory programming courses. In particular, object-oriented programming is often considered more problematic, given that its concepts are tightly interrelated and cannot be easily taught and learned in isolation [1].

In this paper, we present a tool consisting of an interactive environment for experimenting object-oriented Java programs that we refer to as AGUIA/J[1]. Such a tool is an adaptive application that takes as input classes developed by the user and adapts itself according to those classes, providing a graphical user interface to create and control objects of those classes. Within the tool, the objects "gain life" in their own widgets, which a user may interact with by invoking operations and observe how the objects' state evolve. When recompiling the classes, the user may request the system to adapt at runtime, implying the existing objects to instantly morph according to the new versions of the classes.

Our approach is centered on having programming language features metaphorically represented by the GUI elements of the tool's user interface. The object widgets take different forms according to their structural members and state. For instance, operations are mapped to buttons and are invoked when the button is pressed, different data types have different kinds of widgets exhibiting them (e.g. a boolean is represented as a check box, while an enum type is represented as a set of radio buttons), private class members are not displayed, etc.

We have conducted an experiment using AGUIA/J in an introduction to programming course. The experiment was based on having pilot lab class groups using AGUIA/J as courseware. The results have shown that the lab groups where AGUIA/J was used registered significantly higher approval rates and lower drop-out rates.

This paper proceeds as follows. Section 2 briefly presents the philosophy underlying the proposed tool and the requirements that drove its elaboration. Section 3 presents the AGUIA/J tool resorting to concrete usage examples. Section 4 presents the evaluation of the tool, detailing the experiment and discussing its results. Section 5 situates our approach regarding related work, and finally, Section 6 concludes the paper.

2. APPROACH

Nowadays it is common to adopt an object-oriented language in introductory programming courses. However, these courses typically suffer from high drop-out and failures rates. Our approach consists of a new tactic for understanding object orientation backed up by tool support. We aimed at a tool that was adequate as courseware of an introductory programming course where Java is taught. We chose the Java language given its widespread use in teaching.

Our approach is strongly based on drawing *conceptual metaphors* from the source domain of GUI elements to the target domain of object-oriented concepts. A conceptual metaphor is a metaphor in which one concept is understood in terms of another concept. The conceptual domain from which metaphors are drawn to understand another conceptual domain is known as the *source domain*, whereas the conceptual domain that is understood in this way is the *target domain*. Conceptual metaphors typically employ more abstract concepts as their targets and more concrete concepts as their sources.

In our approach, the metaphors serve the purpose of illustrating object-oriented concepts within a graphical environment. The ad-

[1]Adaptive and Generic User Interface Application (for Java)

ITiCSE'11, June 27–29, 2011, Darmstadt, Germany.

```
public class Point {
 public double x, y;

 public Point(double x, double y) {
  this.x = x;
  this.y = y;
 }

 public static Point origin() {
  return new Point(0.0, 0.0);
 }

 public void move(double x, double y) {
  this.x += x;
  this.y += y;
 }

 public boolean isOrigin() {
  return x == 0.0 && y == 0.0;
 }

 public boolean isEqualTo(Point p) {
  return x == p.x && y == p.y;
 }
}
```

```
public class Line {
 public Point start, end;

 public Line(Point start, Point end) {
  this.start = start;
  this.end = end;
 }
}
```

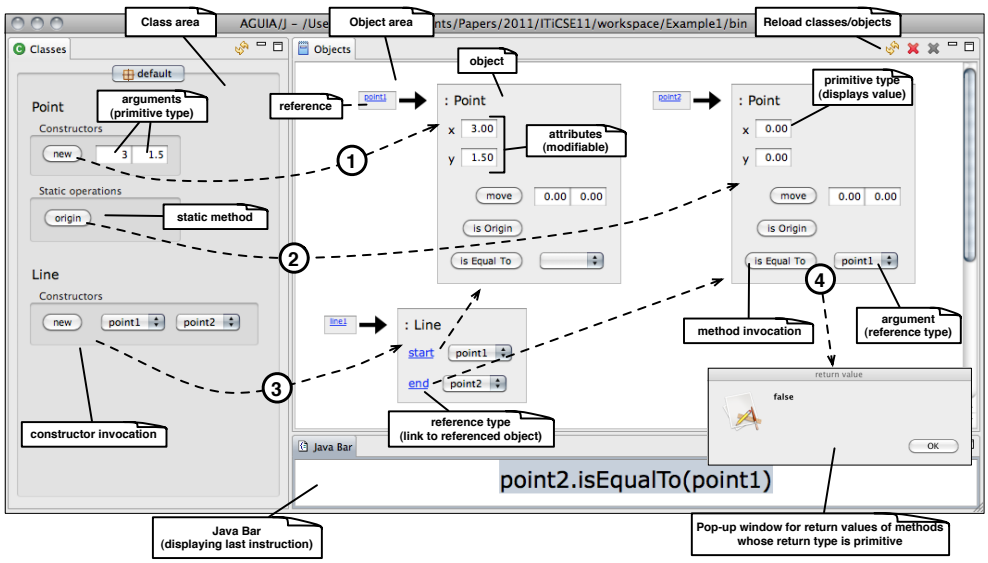

Figure 1: Objects and references in AGUIA/J.

vantage of using a source domain of GUI elements for the metaphors is that it is very well-known, simple, and with a visual representation. A central metaphor that we represent in the tool is "object as machine", inspired on [5] (Figure 7.5). The idea is to shape apprentices' perception of objects as machines with several displays (properties) that may change when interacting with the machine by pressing its buttons (operations).

The machine is like a black box that only shows the external interface of the object. The user is allowed to interact with the objects according to what could be done programmatically through its public interface. This issue is closely related with the perception of encapsulation, and we believe this to be a powerful metaphor for grasping such a concept.

As argued in [3], we believe that starting to exemplify Java with its "main" method is pedagogically incorrect. The reason is that the "main" method has nothing to do with object orientation, and thus, we are starting with an exception to the rule. Moreover, early introduction of language features related to I/O can be considered harmful. The reason is that it detracts apprentices from the essential concepts and forces teachers to "hand wave" when addressing necessary language features that involve more advanced topics (e.g. exceptions). These issues drove the design of the BlueJ environment [2]. Our approach follows this philosophy, enabling apprentices to exercise the several object-oriented concepts without resorting to the "main" method or any language features related to I/O.

In order to achieve enhanced usability and interactivity, it was our goal to support the capability of runtime adaptation upon recompilation of user classes. This means that a user may be experimenting some objects and on request morph the existing objects according to the new version of their classes. This enables an agile switching between concept explanation and exemplification. Changing the type of attributes, introducing new methods, changing methods to static, or changing access modifiers, are examples of modifications that can be performed on the classes and after which the objects in the environment adapt according to such changes.

3. AGUIA/J

This section presents AGUIA/J, a tool we have developed according to the philosophy and goals described in the previous section. AGUIA/J is a stand-alone GUI application built using Java SWT that looks as shown in the right-hand side of Figure 1.

The AGUIA/J window is divided into a class area (left) and an object area (right). The class area adapts itself according to the set of classes and their constructors, whereas the object area is populated as the user interacts with the application. The object area displays widgets that represent "objects as machines", which may be controlled by the user. On the bottom part we can find the Java Bar, which displays the Java instructions that are equivalent to the user actions performed through the user interface (e.g. object creation, method invocation). Alternatively to controlling the objects directly, the user may type Java instructions to create and invoke operations on the objects.

In order to use the tool, the user develops simple classes without any instantiation code (i.e. the "main" method) or additional configurations, and further launches the tool passing the working directory where the compiled class files are located. The tool will adapt itself according to the classes (and packages) found at the given directory. On request, the user may reload the environment with new versions of the classes without having to restart execution. The existing objects that were created remain on the environment and morph according to the new versions of their classes.

AGUIA/J executes user code in a "sandbox" so that runtime errors and infinite cycle bugs are captured and presented to the user in a friendly manner. In case of a runtime exception caused by user code (e.g. NullPointerException, ArrayOutOfBoundsException), an error dialog will pop-up, containing a more apprentice-friendly error message that indicates the exception (in contrast to a stack trace), as well as the location in the code where it occurred. With respect to infinite cycle bugs, AGUIA/J has a configurable time-out parameter that sets the number of seconds that the execution of a method/constructor is allowed to last, until execution is interrupted and a warning message informing about a possible infinite cycle bug is displayed.

```
public class Point {
 private double x, y;

 public Point(double x, double y) {
  this.x = x;
  this.y = y;
 }

 public String toString() {
  return "(" + x + ", " + y + ")";
 }

 public boolean isOrigin() {
  return x == 0.0 && y == 0.0;
 }

 public double getAbscissa() {
  return x;
 }

 public double getOrdinate() {
  return y;
 }
}
```

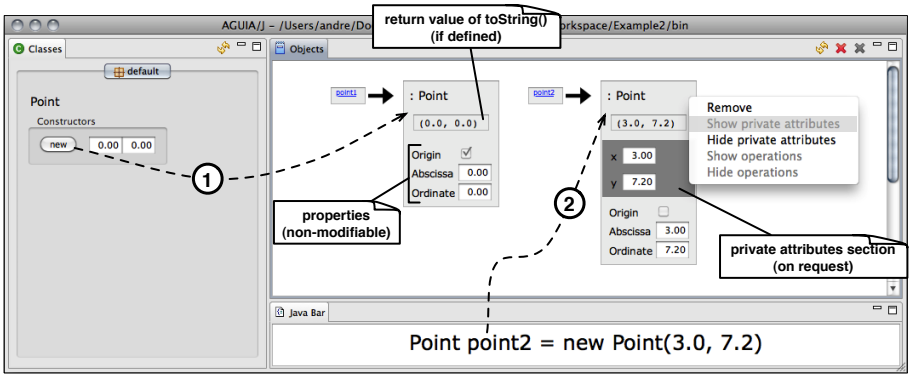

Figure 2: Encapsulation and properties in AGUIA/J.

The next subsections illustrate AGUIA/J's features with examples. AGUIA/J covers many other features that are not illustrated in this paper due to space constraints. With respect to object-oriented concepts, these features essentially comprise inheritance, polymorphism, and enumerations.

3.1 Objects and references

As explained, the tool adapts itself to a set of classes. The first step of such adaptation is related to the class area on the left-hand side of the application window (see Figure 1). This area will contain a section for each class defined according to the different ways of creating objects of such class. These are determined by the existing public constructors. For each constructor there will be a button for creating an object using that constructor. The created objects will appear in the object area on the right-hand side of the application window.

After being created, each object is displayed in a widget with several sections. In the objects of Figure 1 we can see a header with the object's type, a section with the public attributes, and a section with the public operations for that object. An object widget enables interaction with the object through its external interface, as if the object would be used programmatically. Both static methods and static attributes of a class appear on the left-hand area rather than on each object, in order to give the intuition that they are not associated to any particular object.

Figure 1 presents two simple classes, Point and Line, whose members are all public. The screenshot of the tool is annotated with the interaction steps that were taken before the reaching the shown execution state. Although having non-private attributes is not a best-practice in Java, enabling attributes to be visualized is useful to transmit the basic notion that objects have state. We believe it is beneficial to be able to delay the use of access modifiers until the concept of encapsulation is introduced.

On step one, a Point object was created by typing the necessary arguments in the constructor widget and clicking the "new" button. Upon doing so the newly created object appears on the object area, as well as a reference to it. Reference names are given automatically by the tool. This decision favored usability over functionality, given that asking an identifier to the user upon the creation of each object would slightly slow down interaction. However, if using the Java Bar the user may choose which reference names to use. On step two, the static method Point.origin() was invoked. Because the method returned an object, such object was placed on the object area. On step three, a Line object was created passing references point1 and point2 (which were assigned to the previously created Point objects). Arguments of a reference type will have a drop-down list from which an existing compatible reference can be selected. References are represented by links, on which a user may click to obtain a new reference to the referenced object. The state of the objects is always kept updated with the current value of the objects' attributes. Operations may be invoked on an object by clicking on its buttons. For instance, on step four, the operation isEqualTo(Point) was invoked. Since its return value is of a primitive type, a dialog window pops-up displaying the returned value.

3.2 Encapsulation and properties

Encapsulation is an important concept in object orientation that often apprentices fail to grasp. In Java, encapsulation is achieved using access modifiers (i.e. public, protected, default, and private). In AGUIA/J, the possible interactions with the objects are determined by their external interface. In this way, the "machine" that represents the object only enables the user to visualize and interact with the public members of the object. Therefore, non-public attributes are not displayed and non-public operations are not available for invocation, just as if the object was being used programmatically. We believe that this mechanism is useful to help on developing the intuition that encapsulation is a means to control how an object can be used and to separate interface from implementation. For debug purposes, the user may activate a section of the object that displays its private attributes (see Figure 2).

The *command query separation* principle [5] states that a method should either implement a *query* that does not change the object's state or a *command* that possibly changes the object's state, not both simultaneously. The return values of query methods that require no arguments can be thought of as being object *properties*, which are either calculated or directly given by attribute values. Based on the method signatures and on the widely-used naming conventions (*get**, *is**), AGUIA/J detects which methods are representing property accessors. In this way, for each method considered as a property accessor there will be a field showing its return value at every moment of execution (see Figure 2). Naturally, the fields that exhibit properties are non-modifiable. In case a method detected as a property accessor modifies the object's state, the tool displays a warning. Although what described is the default tool behavior, it is possible to customize the property detection policy (or disable it).

Finally, if a class overrides Java's toString() method, its return value is handled as a special property, appearing in a dedicated section on the top of the object widget (see Figure 2).

3.3 Arrays

AGUIA/J features array visualization and interaction, representing them as objects with a field for each array position. Figure 3 illustrates arrays resorting to the class Point given previously. On step one, an array of type Point was created using the JavaBar. On step two, a Point object was created. On step three, the Point object was assigned to the first position of the array through the drop-down list (which includes the references with a compatible type). Finally on step four, the Point object is obtained by clicking on the link of the first position of the array.

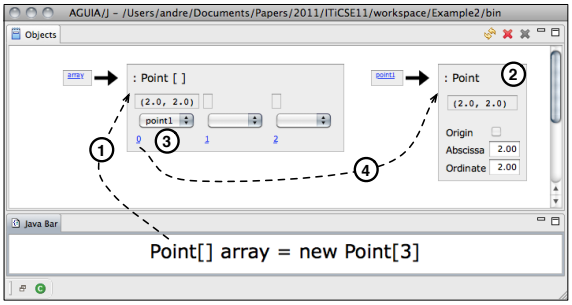

Figure 3: Arrays in AGUIA/J.

4. EVALUATION

In order to evaluate AGUIA/J we have conducted an experiment involving the introductory programming course taught at our department. This section details such an experiment and presents its results concerning drop-out and approval rates.

4.1 Context

The introduction to programming course duration is 12 weeks, each one having a lecture of 1,5 hour and a lab class of 3 hours. On a given week, the lecture covers certain topics, whereas the lab class is based on a set of exercises related to those topics. Lab classes are designed to have an initial summary, where the teacher addresses the class, but, during the remainder of the class the teacher helps each student individually. University policy requires every class (both lectures and lab classes) to have a record of attendance.

During a semester there are several lab class groups working separately. The number of students in a lab group is typically between 20 and 30. In order to obtain approval in the course, a student has to perform the final exam. Attending the final exam requires the student to obtain prior approval in the lab classes, by means of tests and assignments.

The course has roughly 3 stages, which are summarized in Table 1. In past course editions, records show a considerably high drop-out between the end of Stage 2 and the beginning of Stage 3. Typically, if a student reaches Stage 3 with no reasonable skills regarding the previous topics, either gives up the course or proceeds but fails to obtain approval. In this course, a drop-out rate of more than 30% is often common.

Table 1: Introduction to programming course stages.

Stage	Weeks	Topics
1	4	Variables, loops, selection statements
2	4	Arrays, references, functions/procedures
3	4	Classes, encapsulation, enumerations

4.2 Experiment

The experiment involved two of the degrees where the course is taught, namely *Computer Science and Engineering* (CSE) and *Informatics and Management* (IM). On the semester when the experiment was carried out, each of the degrees had 4 lab groups, totalizing 8 lab groups. The lab groups registered an initial number of students of approximately 85 in CSE and 110 in IM. The experiment took place in 2 out of the 8 lab groups, having one pilot lab group from each degree. The other 6 lab groups had the role of control groups in this study. The distribution of students by the lab groups was performed by the university in the normal manner, without our intervention. The author of this paper was the teacher in the 2 pilot lab groups, while the other 6 control lab groups were taught by 5 different teachers. The author was neither involved in the elaboration of the final exam nor on its grading.

The experiment consisted of providing the pilot groups with AGUIA/J, so that the students would perform the lab exercises using it. The lab exercises for the pilot groups were the same as in the control groups (i.e. the regular course exercises). The students had approximately 8 hours of contact with the tool, spanned across 4 lab classes on the Stage 3 of the course. We are aware that some of these students have used the tool outside the class, but we have no concrete data for quantifying this factor.

Slightly after the middle of the semester (on week 8) there was a small compulsory test that every student had to do in the lab class as part of the course evaluation. Given that this test took place just before the experiment of using AGUIA/J has started, it was a good indicator of how the student population was distributed among control and pilot groups. Table 2 presents information about the test, namely attendance distribution among pitot and control groups (*dist*), average score (*avg*), and standard deviation of scores (*std*). We can see that in CSE the scores are similar, while in IM the average score is a little lower, but with less disperse scores. In CSE the number of students of the pilot group that performed the test was a quarter of the total (25%), whereas in IM the pilot group represented slightly over a quarter of the total (28%).

Table 2: Mid-semester test: pilot and control group results.

Group	CSE			IM		
	dist.	avg.	std.	dist.	avg.	std.
Pilot (1)	20 (25%)	73%	20%	28 (28%)	63%	18%
Control (3)	59 (75%)	73%	25%	72 (72%)	69%	27%

4.3 Results

With respect to exam attendance (see Figure 4), which was only possible for students that succeed in the lab class evaluation, we observed that, for both degrees, the proportion of pilot group students was higher than the proportion of pilot group students that attended the mid-semester test. This means that the pilot groups as a whole were more successful in reaching the exam than the control groups. Moreover, the proportion of pilot group students that obtained approval in the course was higher than the proportion of pilot group students that attended the exam. This means that the pilot groups as a whole were more successful in the exam than the control groups. Figure 4 also presents a chart with the exam success rates, according to degree. Here it is only considered the students that have made it to the exam, thus excluding drop-outs.

We believe that these results provide evidence that the usage of AGUIA/J as courseware was effective. However, as a threat to validity, we acknowledge that the fact of having different teachers on pilot and control groups could have influenced the results. On the other hand, the fact that the teacher of the pilot groups was not in-

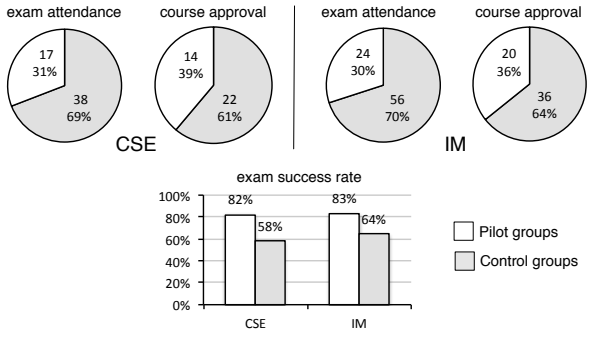

Figure 4: Exam attendance and course approvals.

volved in exam elaboration and grading strengthens the impartial nature of the measurements that determined the results.

Finally, we have analyzed the records of student attendance in the lab classes. Having the initial number of students that was attending a certain lab class and the number of students that remained until the last week, we calculated the drop-out rate for each lab group. Figure 5 presents a chart with the course drop-out rates, according to degree. We can observe considerably lower drop-out rates in the pilot groups of both degrees. The higher success of the pilot groups in reaching the exam is in principle related with this indicator, given that if more students remain in the course, there will probably be more students reaching the exam.

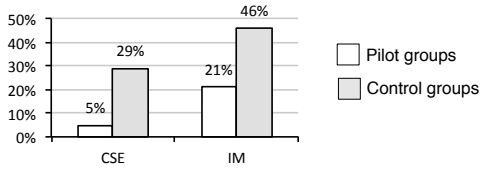

Figure 5: Drop-out rates.

5. RELATED WORK

BlueJ [2] is an established and popular environment for teaching and learning object orientation using Java. AGUIA/J has several aspects in common with BlueJ. Both tools have a code pad where users may type in instructions to create and interact with objects. Another common aspect is the fact that apprentices neither need to develop I/O code nor to develop the "main" method in order to instantiate object classes. Despite the similarities with BlueJ, AGUIA/J has some fundamental differences with respect to the representation of objects, interaction style, and tool usability.

A fundamental difference pertains to the fact that in BlueJ the object fields are displayed homogeneously in a textbox using their textual value, while in AGUIA/J the fields assume different widgets according to their type, access modifiers, and state. Moreover, AGUIA/J features a graphical metaphor for encapsulation, whereas for instance in BlueJ the difference between the representation of a private and a public attribute is solely textual.

Upon the creation of an object, BlueJ displays on its object bench a red block that represents a reference to the created object. In AGUIA/J, the user is always faced with the actual created object, which later on might have several references pointing to it. This multiple-to-one relationship is not made explicit in BlueJ, since distinct reference red blocks do not provide any graphical hint in case they are pointing to a same object.

Finally, AGUIA/J is capable of readapting itself on request to new class versions, enabling the user to see the existing objects in the environment morphing according to the version of the classes. While in AGUIA/J the user may continue interacting with the existing objects, in BlueJ the existing objects in the environment disappear upon recompilation of their classes.

DrJava [6] is a pedagogical programming environment where apprentices develop Java classes and input Java expressions in an interactive console for experimenting objects based on a "read-eval-print loop". As in AGUIA/J, and BlueJ, DrJava also relieves apprentices of dealing with I/O. In contrast to AGUIA/J, DrJava does not attempt to draw graphical conceptual metaphors for understanding object orientation.

Jeliot [4] is a pedagogical animation system for Java programs. Such a system is capable of animating a Java program from beginning to end in a friendly and intuitive environment. However, Jeliot does not allow interactive experimentation of objects as in AGUIA/J, BlueJ, or DrJava. Therefore, despite the fact that apprentices may visualize their programs in Jeliot, the tool is not the best choice for students to develop lab class exercises, as suggested by its authors. Moreover, given that in Jeliot all classes have to be given in a single file and that every object creation is visualized, the tool usage does not scale well for large examples.

6. CONCLUSION

In this paper we proposed a tool (AGUIA/J) that exploits graphical conceptual metaphors based on GUI elements for addressing the problem of understanding of object orientation. The approach was evaluated by accessing approval and drop-out rates of pilot lab groups that were using AGUIA/J as courseware. The main innovative aspects of the tool comprise (a) representing "objects as machines" according to its structural members and state, (b) a mechanism for addressing the query-command separation principle based on object properties, and (c) the capability of runtime adaption to new class versions.

7. ACKNOWLEDGMENTS

I would like to thank Luís Nunes for his valuable comments. This work was partially supported by FCT through LASIGE (U.408) Multiannual Funding.

8. REFERENCES

[1] J. Bennedsen, M. E. Caspersen, and M. Kölling. *Reflections on the Teaching of Programming: Methods and Implementations*. Springer Publishing Company, Incorporated, 2008.

[2] M. Kölling, B. Quig, A. Patterson, and J. Rosenberg. The BlueJ system and its pedagogy. *Journal of Computer Science Education, Special issue on Learning and Teaching Object Technology*, 13(4):249–268, December 2003.

[3] M. Kölling and J. Rosenberg. Guidelines for teaching object orientation with java. In *ITiCSE '01: Proceedings of the 6th annual conference on Innovation and technology in computer science education*, pages 33–36, New York, NY, USA, 2001.

[4] R. B.-B. Levy, M. Ben-Ari, and P. A. Uronen. The Jeliot 2000 program animation system. *Comput. Educ.*, 40(1):1–15, 2003.

[5] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[6] E. A. Robert, R. Cartwright, and B. Stoler. Drjava: A lightweight pedagogic environment for java. In *SIGCSE Bulletin and Proceedings*, pages 137–141. ACM Press, 2002.