An Open-Ended Environment for Teaching Java in Context

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

Teaching programming in context, i.e. having students learning how to program by manipulating artifacts of a familiar domain (e.g. images, card games), has demonstrated convincing results in terms of raising student retention and interest in CS. This paper presents a pedagogical environment for teaching Java in context that is extensible with respect to the domains it supports. Instructors model domains and develop visualization widgets for rendering their objects. In turn, students use the environment to visualize and manipulate objects of the domain when solving exercises. The advantage and originality of the proposed environment is that it embodies an open-ended platform for creating diverse contexts at a low cost, standing as an enabler for widening the spectrum of abstractions for programming in context.

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

Keywords

Object-oriented programming, contextualized programming, Java

1. INTRODUCTION

When teaching computer science in *context* [2], programming is introduced using domains that are familiar and relevant to students, such as image manipulation or card games. This teaching strategy has proven successful (e.g. [9]), especially in non-CS majors (e.g. [4]), demonstrating a significant impact on retention and student satisfaction.

This paper presents an extensible environment[1] for teaching and learning Java in context. When using the environment, instructors may adopt existing or create new *domain contexts* for programming exercises. Such domain contexts may be, for instance, board games, card games, image manipulation, charts, maps, etc, and basically anything that can be modeled and visualized. In order to supply the environment with a domain context, one has to provide a set of classes that models that domain and view widgets that are capable of rendering objects of those classes.

When developing and testing programs, students manipulate and visualize objects of the domains provided by the extensions available in the environment. In a first step, a student may explore a domain in order to learn what can be done with its objects. This is achieved by interactively creating and manipulating objects through the environment's GUI. Upon understanding the domain, students instantiate its classes in their programs when solving exercises. The role of the environment is to enable experimentation of the developed code, through a GUI that adapts itself to the user classes.

The advantage of the proposed environment is that it embodies an open-ended means to support various domains for programming in context. The environment extensibility allows domain contexts to be added at a low cost, in the order of a few tens of lines of code. Standing as an affordable means to develop new contexts, the environment has the potential for widening the spectrum of solutions for contextualized programming.

The proposed environment has been used during one delivery of the introduction to programming course at our department. We have carefully analyzed its impact with respect to two different student populations (CS majors and non-CS majors). Results reveal higher success rates in both populations, however, with more expressive impact on non-CS majors. Moreover, we found that non-CS majors tend to like and feel more comfortable with the environment than CS majors.

This paper proceeds as follows. Section 2 presents an overview on the proposed environment. Section 3 explains how the environment can be used by students. Section 4 details how extensions for supporting new domains can be developed. Section 5 presents the results of using the proposed environment in our introduction to programming course. Section 6 situates our approach regarding related work, and Section 7 concludes the paper.

2. OVERVIEW

The proposed environment is an Eclipse-based tool supporting Java that capitalizes on our previous work on AGUIA/J [8]. The essence of the environment's role is to graphically render objects of a certain domain context, while enabling users to manipulate them. The domain contexts supported by the environment are provided by means of plugins. In this way, the environment is not targeting any particular domain, but it rather consists of an open-ended platform for supporting different domains, which may be combined.

[1]available at *http://www.aguiaj.org.pt*

ITiCSE'12, July 3–5, 2012, Haifa, Israel.

For the purpose of illustrating our approach, consider a basic domain context of board games. Many board games involve a rectangular board, composed of several positions that are referenced by a coordinate (line and column). Positions may have different colors and may hold an image (icon) that represents a game entity (e.g. a chess piece). Figure 1 presents a conceptual model for board games according to what is described, and a picture representing an instance of that model.

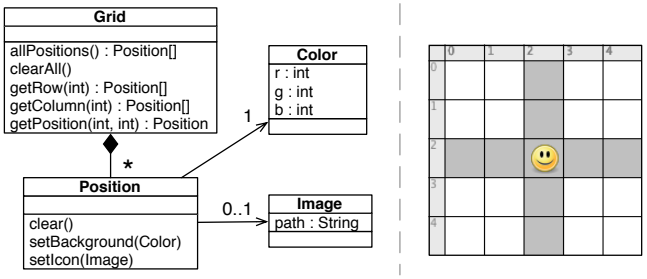

Figure 1: Board games context: a UML class diagram (left) describing the domain model and a picture (right) representing an example instance of the model.

We refer to the objects that instantiate the domain classes as being a *model* of a particular board in a certain state, whereas we refer to the several possible visual representations of that model as its *views*, as in the the well-known *model-view-controller* architectural pattern. For instance, the graphical representation of the board shown in Figure 1 is one out of many possible representations, from which the actual *domain model* is independent. When developing a domain context for our environment, one has to model it in terms of Java classes.

The environment customization relies on a plugin-based architecture, which we illustrate in Figure 2. A plugin is essentially composed of a *domain model* and *view* widgets. The domain model is given as a set of classes, whereas the views are widget implementations for rendering objects of those classes. The environment acts as a *controller* that coordinates the domain objects and the views that render them. There are two clearly distinct environment stakeholders. On the one hand we have *instructors*, which prepare the environment to be used in accordance with the proposed course exercises. On the other hand we have *students*, which use the environment as a learning tool and solve exercises using it.

The role of an instructor is to customize the environment for a given programming course (or a course involving programming). The customization can be achieved by means of: (a) using existing plugins that address domain contexts, (b) developing a plugin for supporting a new domain context, or (c) adapting/extending an existing plugin. Naturally, the proposed course exercises have to take into account the domain contexts available in the customized environment.

In turn, students solve exercises involving the creation and manipulation of objects of the domain contexts offered by the environment. In a first contact with a new domain, a student may create its objects solely through the environment's GUI, in order to easily see what the objects are representing and understand their behavior in terms of the available operations. Upon a domain being understood, students may start to manipulate the domain objects in their Java programs. Furthermore, if inheritance is a topic of the course contents, the exercises may involve specialization of domain classes.

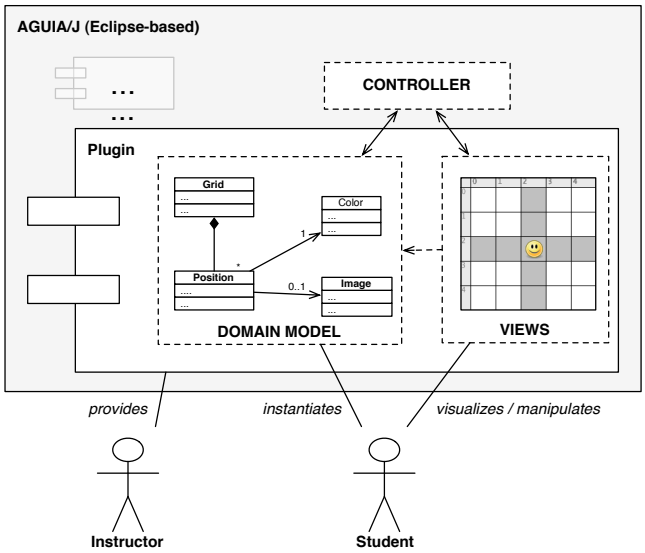

Figure 2: Environment overview: instructor and student roles.

3. ENVIRONMENT USAGE

This section explains how students use the environment, detailing the user experience when solving programming exercises.

3.1 Domain exploration

As explained, a domain is modeled in a set of classes. These classes conceptually define the set of possible objects of the domain. When faced with an unfamiliar domain, a student may explore it through experimentation in order to understand its objects. Domain objects are created and manipulated in the environment, enabling the user to see the changes in the objects' state while interacting with them. This is illustrated in Figure 3 using the board games domain introduced previously. The boxes represent the objects and the buttons represent commands (i.e. operations) that can be instructed to the object (by pressing them). While interacting with the objects, the user visualizes their state in a view widget on the upper part. Following the annotations in the figure, in step one the Grid object was created by pressing the button new. Upon creation, all the positions of the grid were white. In step two, the method getPosition(..) was invoked on the grid, causing the corresponding position object to be brought into the environment. Finally, in step three, the method setBackground(..) is invoked on the position fetched in step two, changing its color to gray. The color changes in both widgets (grid and position), since we have the position object visualized in two different views.

During interaction, the environment displays a documentation view that shows the relevant Javadoc snippets when the mouse hovers class/object members. For instance, when the mouse is over an object's button, the documentation view displays the Javadoc description of that operation (as illustrated in the figure).

3.2 Domain instantiation

After gaining contact with a given domain context, students will have grasped what can be done with the domain objects. This will in principle make it easier to instantiate the domain classes in Java programs, given that the objects should feel less abstract after being materialized in the environment.

In order to use a domain context, its classes have to be imported as if using a library. The output of the exercises relies on the envi-

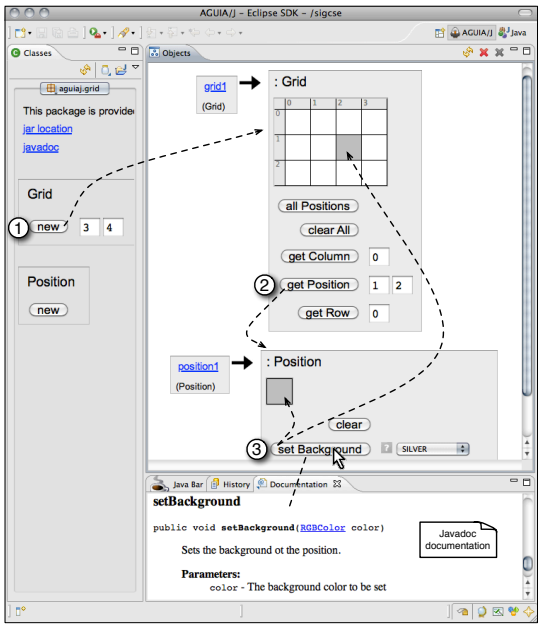

Figure 3: Domain exploration: platform screenshot illustrating the experimentation of the board games domain.

```
1  import aguiaj.board.Grid;
2  import aguiaj.board.Position;
3  import aguiaj.colors.Color;
4
5  public class CheckersGame {
6     private static final int SIDE = 8;
7     private static final Color BLACK = new Color(0, 0, 0);
8
9     private Grid board;
10    ...
11    public CheckersGame() {
12       board = createBoard();
13    }
14
15    public static Grid createBoard() {
16       Grid board = new Grid(SIDE, SIDE);
17       for(int i = 0; i < SIDE; i++) {
18          for(int j = (i+1) % 2; j < SIDE; j+=2) {
19             Position p = board.getPosition(i, j);
20             p.setBackground(BLACK);
21          }
22       }
23       return board;
24    }
25
26    public Grid getBoard() { return board; }
27    public void setupGame() { ... }
28    ...
29 }
```

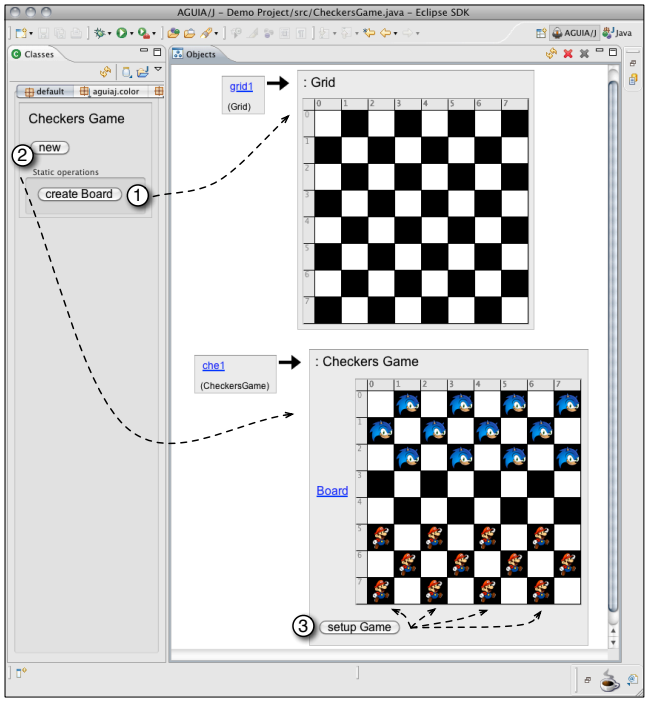

Figure 4: Domain instantiation: using the board games domain to develop a checkers game.

ronment, which is where users see the objects that their code outputs. The environment adapts its GUI to the structural elements of the user classes.

As an example, Figure 4 presents a part of an exercise for creating a checkers game using the board games' domain classes, omitting attributes and methods related with the game state. The class CheckersGame is the main class of the game, holding a reference to a Grid object, which displays the board view. In this example we can see two forms of experiencing the visualization of domain objects in the environment. The first form is direct object instantiation, illustrated with the static method given on lines 15-24. Given that the method is public, a button for invoking it is displayed in the GUI. When pressing it (step one), the method is invoked and the resulting Grid object is placed in the environment. The second form is when domain objects are part of objects of a user class, as illustrated with CheckersGame. Such a class has a constructor, which determines the existence of the new button. When pressed (step two), the created object of type CheckersGame is brought into the environment. Due to the existence of the accessor method getBoard(), the object displays the current game board using the available widget to render Grid objects. In step three, the method setupGame() is invoked by pressing the button, setting up the initial state of the game.

4. PROVIDING DOMAIN CONTEXTS

Providing a domain context for the environment can be achieved by means of developing a plugin. Given that the plugin-based architecture of the environment is built on Eclipse, the extension points are based on its standard mechanisms. Although we have assumed that plugins would be developed by instructors or anyone with a reasonable programming background, we consider that the necessary programming skills for doing so are not demanding.

A plugin has to provide a *model* for the domain and *view* widgets for rendering objects of that domain. The model is composed of a set of classes, which are completely independent from environment classes or any other classes (such as Swing/AWT). Therefore, the development of the model does not involve subclassing environment classes nor implementation of interfaces. The domain should be modeled abstractly and independently of how its objects might be visualized (recall Figure 1). The domain classes are intended to be a "pure" and conceptually clean model of the domain that one wants to support in the environment. Given that domain classes are going to be used and inspected by students, their code should be clean and without any awkward workarounds that could be considered pedagogically incorrect. Finally, the code should include Javadoc comments, so that students can have access to documentation when exploring the domain (recall Section 3.1). The effort of creating the domain model is orthogonal to our approach, given that it directly relates to the complexity of the domain itself.

In addition to the domain model, a plugin should provide view widgets that are responsible for rendering the domain objects. The

widget classes have to implement an interface of the environment's framework. The classes of the view widgets depend on the model classes (while the opposite should not happen). Given that Eclipse is built on SWT, the view widgets have to be implemented using it. As a reference in terms of effort, the implementation of the grid viewer that was used throughout the examples in the previous section had approximately 150 LOC. This domain also included a view widget for the position in isolation (as in Figure 3), which had approximately 60 LOC.

4.1 Card games domain context

In order to give another example of a domain context, here we present a domain context for *card games* that we developed for the introduction to programming course at our institution. Besides having course exercises making use of this domain, students also had to develop a small project using it, as part of their evaluation.

Card games are known almost everywhere in the world and are very popular in some countries. Possible programming exercises span across the domain of known card games. Algorithmic topics involving sorting, shuffling, randomness, and array manipulation, naturally fit within this context. Figure 5 illustrates the playing cards domain by presenting a screenshot taken from a student project (object visualized in the environment) and the related domain model. As we can see in the domain model, the concepts of *rank* and *suit* were represented as object enumerations. Besides the concept of card object, the domain also considered data structures for cards, namely, for representing *hands* and *stacks* of cards.

The implementation of the card views totalized approximately 60 LOC, comprising two widgets (for cards and suits). The card stack widget had approximately 50 LOC, while the card hand widget had approximately 70 LOC.

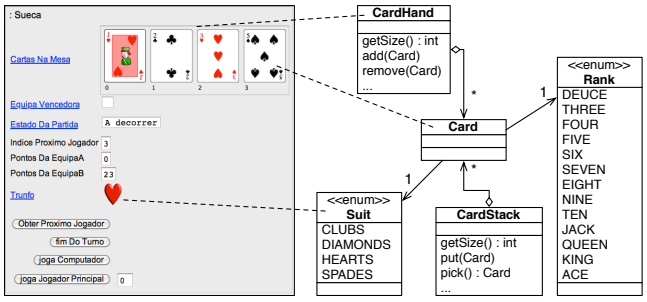

Figure 5: Playing cards context: Screenshot of a student project and domain model.

5. EVALUATION

At our institution, we have redesigned the introduction to programming course taking into account the proposed environment, while shifting to a conservative contextualized programming teaching strategy. By conservative we mean that the course still remains partially traditional. In our case, the first half of the course is taught using traditional exercises that mostly involve numeric calculations and algebra, whereas in the second half the different domain contexts come into play.

5.1 Context

The introduction to programming course is offered to several degrees, which are all related to Information Technology. In this section we report the results of the last delivery of the course for two of those degrees, namely Computer Science and Engineering (CSE, CS major) and Informatics and Management (IM, non-CS major).

The course lasts for 12 weeks, each week having a lecture of 1.5 hour and a lab class of 3 hours. Table 1 presents the course topics, grouped in three stages of 4 weeks.

Stage	Weeks	Topics
1	4	Variables, loops, selection statements, functions
2	4	Arrays, references, objects, procedures
3	4	Object classes, encapsulation, enum types

Table 1: Introduction to programming course topics.

Each group of students — CSE and IM — has a common lecture given for all the students of that degree and 4 lab class groups running in parallel. The lectures were all given by a single teacher (the main author of this paper), whereas 4 different teachers (including the lecturer) gave one lab class to CSE and one lab class to IM.

Every year, up to 65 new students from each degree enroll in the course. In addition to the students enrolling for the first time there is a significant number of students reenrolling (about 25%).

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 (minimum of 45% score), by means of tests and a small project. Throughout the 12 weeks there were three evaluation items for qualifying for the final exam: (a) Test-1 at week 4, (b) Test-2 at week 8, and (c) individual project to hand in at week 12.

The environment was used throughout the 12 weeks. In the lectures the environment served the purpose of aiding the explanation and illustration of concepts. In the lab classes the environment was used by the students to develop exercises. The environment played a minor role in the first stage of the course, given that students only developed static functions using primitive types. The main advantage was to avoid having to deal with the main() method and I/O.

Besides the lab class exercises, students also used the environment to develop and test their project. The project goal was to model a popular card game using object classes. The game was simulated in an object representing the game state (e.g. see Figure 5). Students tested their card games in the environment and concentrated solely on game logic without worrying about user interface issues, given that the environment works as GUI for their objects. In addition to the card games domain, the course also included exercises using the domains of image manipulation (binary, grayscale, and color) and board games (illustrated in Section 2) in the course exercises.

5.2 Methodology

We decided to investigate and present the results of the degrees CSE and IM in parallel because they had approximately equal population sizes and were taught exactly by the same teachers in equal proportions. Moreover, given that IM is a non-CS major, we could evaluate the impact of the approach on two substantially different student populations.

We have measured the impact of using the environment in terms of approval rates (objective) and student satisfaction (subjective). The evaluation is not only evaluating the impact of the environment itself in isolation, but rather the impact of using the tool in our context together with the course restructuring towards a contextualized programming teaching strategy.

We compared the approval rates among first-time enrollment (i.e. the set of students enrolling for the first time in the course) between the restructured course delivery and the previous delivery (before adopting the environment). We decided to focus only on first-time

enrollment in order to evaluate the impact of the environment independently of student experience gained in previous enrollments, which naturally influences student performance.

The student satisfaction measurement was achieved by means of a survey handed in to students that have qualified for the exam. We decided to survey only these students in order to guarantee that the respondents had experienced the whole process. One of the survey questions had the purpose of characterizing the population in terms of prior programming experience. The other three questions were related with the way the course was conducted and the involvement of the tool environment in the process. Additionally, in order to complement the comparison of approval rates, the survey included a question only for reenrolling students that asked how difficult they felt the course delivery was when compared to the previous year. The survey was optional and we had 100 respondents, 47 out of 49 in CSE (96%), and 53 out of 55 in IM (95%).

5.3 Results

The two student populations — CSE and IM — are significantly different. Two of the main distinctive characteristics are with respect to prior programming experience and student background. In CSE there are clearly more students with previous programming experience. In IM there are several students with a high school background on Economics and Business Administration. Moreover, the IM degree is more competitive, in the sense that degree enrollment applications normally register a minimum mark that is 5-10% higher than in CSE.

5.3.1 Success rates

Taking into account first-time enrollment only, Table 2 presents how the approval rates evolve as the semester progresses towards the final exam (qualify). The values correspond to the number of students that succeeded in obtaining positive mark (>=50%) on the different evaluation items. We can see that within first-time enrollment roughly two thirds of the students make it to the final exam. We speculate that the remarkable higher success of IM students in the written tests is due to the higher competitiveness.

Table 3 presents the approval rates of first-time enrollment in two course deliveries: *Fall 2010* (previous year) and *Fall 2011* (present year, restructured taking into account the proposed environment). The rates are not definitive, they were originated after the first exam was due (students have a second opportunity). We can observe increased approval rates in both degrees, being the improvement more expressive in IM. We have performed a two proportion Z-test to compare the approval rates of the different course deliveries, revealing a statistically significant ($\alpha = 0.05$) improvement in the Fall 2011 course delivery within IM (p = 0.0436). With respect to CSE, the rate improvement did not prove to be statistically significant (p = 0.2676).

Degree	Enrolled	Test-1	Test-2	Project	Qualify
CSE	62	44 (71%)	39 (63%)	34 (55%)	40 (65%)
IM	59	53 (90%)	45 (76%)	36 (61%)	42 (71%)

Table 2: Evaluation and success rates of first-time enrollment throughout the semester.

Course delivery	CSE	IM
Fall 2010	30.3% (22/66)	33.9% (21/62)
Fall 2011	35.5% (22/62)	49.2% (29/59)

Table 3: Comparison of approval rates of first-time enrollment.

We have asked students reenrolling how they perceive the difficulty of the current course delivery when compared to the previous year. The question in the survey was *"When compared to the previous course delivery, how do you classify the effort required to obtain approval in this course delivery?"*. Within the possible answers: *much easier (-2)*, *easier (-1)*, *equivalent (0)*, *harder (1)*, and *much harder (2)*, we obtained average values of 0.56 for CSE and 0.81 for IM. This poll strengthens the result of improved approval rates, given that a course restructuring that would require less student effort would likely raise approval rates *per se*.

5.3.2 Survey

The survey contained a question that asked students to characterize their programming experience before starting the course (see Table 4). We can see that in CSE there is a significant proportion of students that states that it has considerable prior programming experience, whereas in IM more than 50% of the students stated that they did not have any programming experience at all.

Another question of the survey had the purpose of measuring to what extent students felt that the tool environment helped them (see Table 4). We have segmented students according their answer regarding programming experience ("none" or "little" vs. "fair" or "considerable") and degree. We can see that students that have no significant programming experience tend to answer clearly positively (stating that the environment helped them "somewhat" or "much"), being the tendency stronger in the case of IM (non-CS major). When analyzing students that have stated to have "fair" or "considerable" programming experience we can observe less positive answers. However, in the case of IM there is still 72% of students with programming experience that state that the environment has helped them "somewhat" or "much". It is clearly noticeable that CS majors (CSE) tend to be less positive in their perception of the usefulness of the environment when compared to non-CS majors. It is also worth mentioning that being IM students more positive towards the tool environment they have also been more successful in obtaining approval in the course. Although more CSE students had prior programming experience, their success rate was clearly inferior. Based on comments that we (teachers) heard in the lab classes, we feel that some CSE students tend to not fully enjoy the tool environment due to the frustration of their programming activity not resulting in a "real" program with full control of interaction. This was especially evident in students with prior programming experience.

The survey contained two other questions, which due to space constraints are only reported briefly here. Given that the answers to these two questions did not differ significantly between CSE and IM, we present the results here as a whole. One of such questions was: *"To what extent do you consider the process of developing*

How do you characterize your prior programming experience?

Degree	None	Little	Fair	Considerable
CSE	36%	23%	24%	17%
IM	56%	17%	23%	4%

To what extent do you consider that the tool environment has contributed positively to overcome your difficulties in programming with objects?

Degree	Nothing	Little	Somewhat	Much
CSE	7%	25%	43%	25%
CSE (with experience)	32%	42%	21%	5%
IM	0%	8%	33%	59%
IM (with experience)	7%	21%	43%	29%

Table 4: Survey questions and results.

the project fun (i.e. not tedious, stimulating)?". A great majority of students answered positively, namely 46% answered "very" and 43% answered "fair" (other options: "little" and "nothing"). We concluded that choosing a popular theme (card game) for the project and having the tool environment for developing it was a good option. The other question was: *"Supposing that you have a friend enrolled in a non-CS major degree that has interest in programming, would you recommend him/her this course?"*. A great majority of students answered positively, namely 26% answered "definitely" and 60% answered "yes" (other options: "maybe" and "no"). The goal of this question was to measure to what extent students consider the course to be interesting for non-CS majors.

6. RELATED WORK

The Media Computation [3] approach for introductory programming courses advocates the construction of media through programming, where images, sounds, and videos, are manipulated in students' programs. Our environment, when supplied with a domain context for image manipulation (which may be custom-designed by an institution), is adequate as courseware for a Media Computation course. Moreover, the image manipulation context may coexist with other domain contexts in the environment.

The Inverted Curriculum teaching strategy [7] involves addressing introduction to programming using an "outside-in" approach, where right from the start software construction relies on existing reusable components. These are embodied in a large software framework that the students start to use as consumers, and progressively move on to be producers, exploring, modifying, and augmenting its implementation. The introductory programming course that applies Inverted Curriculum as described by its proponents [7] uses a single, large software framework (named Traffic). Our environment relates to the Inverted Curriculum approach in the sense that students are also provided with a set of classes to work with. Although we have envisioned the domain contexts for our environment as being small, simple, and focused, there is no restriction with respect to their size and complexity. However, using large classes in the domain models would certainly affect environment usability due to the resulting large object widgets.

BlueJ [6] is a pedagogical IDE for Java where users can experiment the creation of objects and method invocations. Despite some usability differences, our environment has a similar interaction style. However, our environment was designed for programming in context upfront. Although BlueJ was not primarily targeted for programming in context, one may use it for that purpose. For instance, the "shapes" example project that ships with BlueJ can be considered to a certain extent a form of contextualized programming, where we have a canvas containing geometrical figures that can be added, moved, etc. However, in this example there is no separation between model and view, since its classes depend on the AWT graphical library (they represent both the concept of geometrical figures and its visualization). Given that one of the central goals of BlueJ is to relieve beginners of having to deal with pedagogically-uninteresting language-specific issues (e.g. main() method), we argue that early exposure to graphical libraries somewhat conflicts with this goal. We are in favor of providing students with conceptually clean models, given that it avoids exposing advanced topics (for a first programming course), while it promotes conceptual modeling (the essence of object-orientation).

Greenfoot [5] is a pedagogical environment for Java (based on BlueJ), suitable for developing programs that consist of 2D simulations in a *scenario* where several *actors* live in a *world*. Greenfoot is extensible with respect to its scenarios, having necessarily every class of visual objects as a subclass of Greenfoot's framework classes (World and Actor). Greenfoot can be thought of as being a means to program in the context of 2D simulations with scenario variability, and naturally, the set of possible domains is constrained to 2D simulations. For instance, it is possible to represent a card game in terms of a world with actors, but nevertheless, it leads to an unnatural modeling of the game.

In a similar vein as Greenfoot, Alice [1] consists of a programming environment, also supporting Java (as of version 3.0), that addresses the domain of creating 3D animations in the style of storytelling. Both Greenfoot and Alice enable students to create animations, a stimulating form of program visualization that we did not consider upfront in our approach. Although these systems may be used in CS1, they were designed to target younger age groups when compared to our approach. We believe that for teaching more advanced concepts, such as algorithms, data structures, and design patterns, the tool support has to move into a more "real" solution.

7. CONCLUSION

In this paper we have proposed an open-ended environment for programming in context whose design considered context variability upfront. An essential aspect of the novelty of our work is the fact that our environment is a means to program in context that is not tied to any particular context. At our institution, we have adopted the environment in the introduction to programming course along with a course restructuring towards a contextualized programming teaching strategy. Results revealed good levels of student satisfaction, as well as a significant improvement in approval rates with respect to non-CS majors.

8. REFERENCES

[1] S. Cooper. The design of alice. *ACM Transactions on Computing Education*, 10:15:1–15:16, November 2010.

[2] S. Cooper and S. Cunningham. Teaching computer science in context. *ACM Inroads*, 1:5–8, March 2010.

[3] M. Guzdial. A media computation course for non-majors. In *Proceedings of the 8th annual conference on Innovation and technology in computer science education*, ITiCSE '03. Thessaloniki, Greece, June 2003.

[4] M. Guzdial and A. Forte. Design process for a non-majors computing course. In *Proceedings of the 36th SIGCSE technical symposium on Computer science education*, SIGCSE '05, St. Louis, Missouri, USA, February 2005.

[5] M. Kölling. The greenfoot programming environment. *ACM Transactions on Computing Education*, 10:14:1–14:21, November 2010.

[6] 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.

[7] M. Pedroni and B. Meyer. The inverted curriculum in practice. In *Proceedings of the 37th SIGCSE technical symposium on Computer science education*, SIGCSE '06, Houston, Texas, USA, March 2006.

[8] A. L. Santos. AGUIA/J: a tool for interactive experimentation of objects. In *Proceedings of the 16th annual conference on Innovation and technology in computer science education*, ITiCSE '11. Darmstadt, Germany, June 2011.

[9] B. Simon, P. Kinnunen, L. Porter, and D. Zazkis. Experience report: CS1 for majors with media computation. In *Proceedings of the 15th annual conference on Innovation and technology in computer science education*, ITiCSE '10, Bilkent, Ankara, Turkey, June 2010.