Teaching OO concepts-a case study using CRC-cards and BlueJ

3 authors, including:

Jürgen Börstler
Blekinge Institute of Technology

136 PUBLICATIONS   **2,034** CITATIONS

Marie Nordström
Umeå University

19 PUBLICATIONS   **193** CITATIONS

TEACHING OO CONCEPTS—A CASE STUDY USING CRC-CARDS AND BLUEJ

Jürgen Börstler[1], Thomas Johansson[1], and Marie Nordström[1]

Abstract — *The transition to object-oriented programming is more than just a matter of programming language. Traditional syllabi fail to teach students the "big picture" and students have difficulties taking advantage of object-oriented concepts. In this paper we present a holistic approach to a CS1 course in Java favouring general object-oriented concepts over the syntactical details of the language. We present goals for designing such a course and a case study showing interesting results.*

Index Terms — *CS1, holistic approach, Java, object-oriented programming.*

INTRODUCTION

The object-oriented paradigm is now taught in most computer science curricula. Most of us experienced that the transition to an object-oriented language in CS1 is more than just a matter of programming language. Switching to object-oriented problem solving and programming has shown to be more difficult than expected. The way we were used to teach introductory programming courses did not work very well for the object-oriented paradigm.

Traditional syllabi for introductory programming tend to focus on syntactical details. Students learn to successively build up larger structures. This works quite well, since everything learned can be reused and reinforced later. The grouping of program elements can be handled as an afterthought. There is no need to introduce high-level and abstract structures, like modules and abstract data types early on. This is very different in the object-oriented paradigm, where several related concepts have to be handled early on (variable, value, type, object, and class).

Concentrating on the notational details of a certain object-oriented programming language prevents students from grasping the "big picture". Most students therefore have difficulties to take advantage of object-oriented concepts. Delivered assignment code is structured even worse than probably would have been using an imperative language.

There are three major problems involved when teaching an introductory course in object-oriented problem solving and programming:

- Lack of suitable and proven pedagogies to teach object oriented concepts and programming.
- Lack of suitable textbooks.

Almost all current textbooks claim to follow an approach particularly suited for object-oriented programming, like "objects-first," "object-centric," etc. However, a close examination of Java textbooks reveals that this can be questioned ([1]).

- Lack of suitable programming environments.

Most environments are too complex for beginners who still struggle with even simple programming tasks.

A HOLISTIC APPROACH

After three years of using Java in our CS1 course(s) we realised the need for a change. Although students acquired sufficient programming skills, we were not satisfied with their mastering of general object-oriented concepts. Our overall goal was to develop an object-centric, holistic approach to the teaching of Java. Objects should be taught from the very beginning with an early focus on design. The teaching of general object-oriented concepts should be favoured over the syntactical details of Java. Misconceptions should be fought as early as possible. We came up with the following rules and guidelines for course design:

No magic. Every topic seeming to involve superior forces (from the student point of view) has to be thought over and rearranged or delayed. Such topics involve for example language idiosyncrasies, like `public static void main (String[] args)` and the usage of overly complex library classes (e.g. input handling) early on.

Do not exemplify with an exception to the general rule. No concepts must be introduced using flawed examples. Java strings for example are not "real" objects, since `String` objects cannot be modified ([9]). The `main` method is not a "real" method, since there are no messages send to it and its parameters seem to be supplied by superior forces.

Objects from the very beginning. Examples involving multiple objects should be used early on.

OOA&D early. Students should be taught a systematic way to develop software given a problem statement. A simple methodology for analysis and design should help them to understand a problem and evaluate alternative solutions *before* starting to code. Doing so early would hopefully prevent them from "hacking" (code first, think later). We wanted to use CRC cards (Class-Responsibility-Collaboration [2]), since we had tried them successfully in other courses.

Exemplary examples. All examples used in classes and exercises should comprise well-designed classes that fill a purpose (apart from exemplifying a certain language specific detail). Examples should be non-trivial and involve multiple classes. All examples should be made available for experimentation.

Hands-on. Topics should be reinforced by means of practical exercises. Lectures should be directly followed up by supervised in-lab sessions. In these sessions students should be walked through successively more complex exercises. For each session, step-by-step instructions and appropriate examples have to be prepared.

Very little "from scratch" development. All software development takes place in context. It is therefore important to be able to read, understand, and modify existing code as well as to develop (re-) usable code. To accept given code is an important lesson to learn for students–Especially for experienced students who often reject "foreign" code and want to have full control over their programs ([4]).

Easy-to-use tools. Students should be provided with tools that support object-oriented thinking. Usability should be favoured before any Java "bells and whistles".

Alternative forms of examination. Examination should not be separated from teaching. The forms of examination should support learning.

Cover basics in algorithm stability. The limitations of computers should be emphasised. Students should learn that computations can produce unexpected results due to limitations in data representation.

Some of these issues have been discussed lively in a recent CACM forum ([5]). Further useful guidelines for Java courses can be found in [6].

COURSE DESCRIPTION

To test our approach we used our Java summer course as a test environment, mainly because summer courses are extra curricula courses, which allow for short-notice changes in contents and design. The schedule and contents of the redesigned course are described in table 1. Entries in the same row were scheduled for the same day. The total workload for the course amounts to four full-time weeks, i.e. students are expected to work about 160 hours for the course.

There are four types of sessions: Traditional lectures in a lecture theatre (L), lab sessions in computer rooms (C), group sessions (G), and quizzes/examinations. The default duration per session is 3 x. 45 minutes (excl. breaks). Reading assignments refer to [7]. For the in-lab sessions we used modified versions of the examples supplied within the BlueJ environment ([3]) and a simplified version of a PacMan like computer game. G1 and G2 refer to our study groups (see section THE STUDY). Activities in G1 and G2 are similar, except where stated explicitly. However, G1 and G2 use different environments from the very beginning with G1 concentrating on object-oriented concepts and design

aspects, whereas G2 focused on Java applications (main method and input handling).

Resource requirements for the course described here are comparable to those before the redesign.

TABLE 1
COURSE SCHEDULE

Sessio	Contents/ Activity	Assignments due
	week 1	
L 1	Introduction to OO; syntax diagrams; using and creating objects; identifiers, variables, and types; references vs. values; assignments and method calls	Read sects 2.1-2.10 (except 2.2,2.4,2.9), 4.1,4.2,1.4,1.5
C 1	Familiarise with environments; copy, compile, and execute code; play around with Shapes example	
L 2	References; classes and methods; base types vs. class types; expressions, operators, and parameters	Read sects 2.4,3.1,3.4,3.5,4.3
C 2	Make well-defined modifications in Shapes example (change/add methods)	
L 3	Control structures; constants	Read rest of sect 3
C 3	Add classes to Shapes example	Assignment 1
Quiz	Quiz; peer marking	
G	G1: OOA/CRC; Intro to PacMan example	
C 4	G2: Strings, I/O; SlotMachine example	
	week 2	
L 4	Inheritance, polymorphism, and dynamic binding; has-a vs. is-a; overriding vs. overloading; UML class diagrams	Read sects 4.4, 7.1-7.5
L 5	Organising code (abstract, interfaces, and packages); method categories; coupling and cohesion; applets vs. applications	Read sects 5.4,7.6, 10.1-10.4 and 0.4,1.4 in [8]
C 5	Define own classes; use inheritance G1: PacMan example G2: Library example	
L 6	Arrays; String class; I/O	Read sects 6.1,6.2,8.4
C 6	G1: Animate PacMan example; output G2: Complete Library example	
L 7	Error handling; files and exceptions; static; Math class	
	week 3	
Exam	Theory exam; peer marking	
L 8	Graphics; layout	Read sects 9.1-9.4
		Assignment 2
L 9	GUIs; event handling; listeners	Read sects 9.5,5.5
	week 4	
L 10	OOA and CRC (G2 only, 2 h) by means of PacMan example	Read OOA/CRC handout
G	CRC session (1 h)	
C 7	Work with various examples	
L 11	Recursion; more on interfaces (sorting objects); numerical accuracy, approximations, and algorithm stability	Read sects 11.3, 11.4 and 12.5 in [8]
C 8	Draw recursive patterns	
		Assignment 3
L 12	Repetition	
Exam	Closed-lab exam (4 h)	

ENVIRONMENTS AND TOOLS

As development environments students could either use the BlueJ development environment or a conventional set-up with an editor and the JDK compiler.

BlueJ is an integrated environment supporting the design, editing, testing, and debugging of Java applications and applets. The most important and distinguishing features of BlueJ are its object-focus and its testing support. The user can compile and instantiate a class directly by clicking on its UML-like representation in the design view. Instances can be inspected and even directly manipulated. Methods may be called and public attributes changed (figure 1). Method parameters are requested on demand by means of an easy to use user interface. It is not necessary to implement drivers, I/O methods, or GUIs. Students can focus on the essential "model" classes of a design. They are not distracted by the supporting classes that make the model classes work, which are difficult to understand at the beginning.
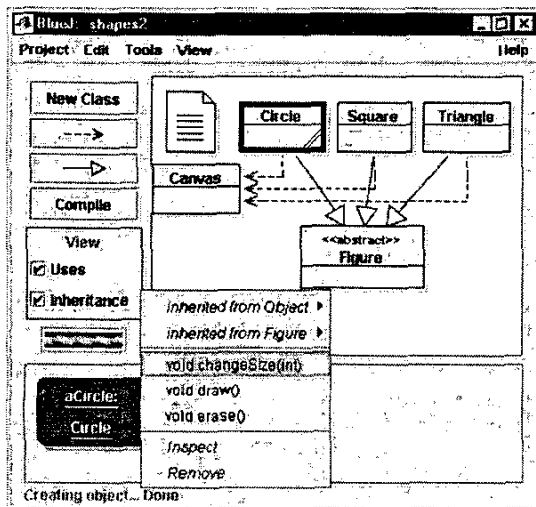

FIGURE 1
BLUEJ WINDOW SHOWING AVAILABLE METHODS FOR A CIRCLE OBJECT

WinEdit is a texteditor for the Windows platform that allows the Java compiler to be called from within the editor. There is a simple form of error tracking, in that the user can step forwards and backward trough the compiler-generated list of errors and jump to the corresponding statements. WinEdit has no project management capabilities.

Both systems are freely available and easy to learn and to use. BlueJ runs on all Java enabled platforms.

To support object-oriented analysis and design we used CRC (Class-Responsibility-Collaboration [2]) cards. CRC cards are a low-tech method to support collaborative problem analysis and design. All objects in a design have a purpose. They take over responsibilities (knowing or doing something) to solve parts of the problem. Complex responsibilities are met by collaborating with other objects. Responsibilities and collaborations are noted on index cards (see figure 2 for an example). The cards are then used to role-play scenarios of system usage. Each student takes on the role of one or more objects and acts only according to the responsibilities stated on his or her card.

FIGURE 2
EXAMPLE OF A CRC CARD FOR A CLASS CIRCLE

THE STUDY

To evaluate the new course design, we conducted a case study during the summer of 2001. The aim was to teach object-oriented programming and problem solving with Java using two different approaches (traditional vs. holistic) in parallel. Two groups of students where monitored closely to see whether we could establish any differences in their view of object-oriented concepts and their ability to develop well-designed solutions.

Due to limited resources lectures had to given jointly to all students. The order of lectures had therefore not been optimised for any of the groups. However during the group sessions (C and G in table 1) we followed two quite different approaches.

- One group (G1) followed an object-centred approach according to the goals of our holistic approach. All examples, exercises, and assignments were based on working examples comprising multiple classes. G1 used BlueJ ([3]) as development environment. Using BlueJ made it possible to concentrate on the basic object-oriented concepts, delaying all "magic" until halfway through the course.

- The other group (G2) was taken down a more traditional path focusing on syntactical details and the development of complete applications from scratch with increasingly complex statements. Students in this group used WinEdit, a conventional Windows editor with some limited JDK integration. I/O was introduced almost immediately. Object oriented analysis and design were not discussed thoroughly until late in the course.

STUDENT DEMOGRAPHICS

Our summer-course is open to all students with at least basic college level mathematics and physics. Student background ranges from CS Masters students with advanced knowledge in ML, C, C++ and Unix to social sciences students with no programming knowledge.

Summer courses tend to have high drop-out rates and low pass/fail ratios (for several reasons that are beyond the scope of this paper). To be able to transfer the results from this study to the courses taught in our CS programs, the

study groups had to be chosen carefully. It was important to get committed students with a background similar to beginners in our program Java courses.

In total 101 students were admitted to the course and 29 volunteered for participation in the study. Based on the results of a questionnaire we selected 13 students for each of the groups. However one (male) student changed groups so we started with 12 students for G1 and 14 students for G2 (see table 2 for details).

TABLE 2
GROUP DEMOGRAPHICS

Question	G1		G2	
	female	male	female	male
No of students	8	4	7	7
Program				
involving CS/IS	8	3	6	6
other	0	1	1	1
Programming experience				
none	1	1	3	0
one language	5	1	4	5
several languages	2	2	0	2
OOA&D experience				
no	2	3	2	3
yes	6	1	5	4
Computer at home				
no	0	0	0	2
yes	8	4	7	5

It can be noted that many students in the study groups are from the universities IS program, where some object-oriented analysis and design is taught before programming. Four students (all female) explicitly noted that their programming experiences were limited, although they indicated having experiences (one of them in several languages!).

DATA GATHERING

To evaluate our course design data were collected from the following events:

- A *quiz* at the end of the first week.

  The students were given 11 questions about syntactical details, parameter passing, and object-oriented concepts. After 40 minutes we presented the results and students marked up their own quizzes on the scale correct, almost correct, partly correct, and wrong/ missing. Each student then compiled all markings into a table and submitted them anonymously (by group).

- A *theoretical examination* halfway through the course.

  This was the main theoretical exam, and replaced the usual exam at the end of a course. The exam was administered in a similar way as the quiz. This time students had to answer 13 questions in three main categories; terminology and syntactical details, the mechanics of programming and understanding of Java code, and questions stressing object oriented concepts in general (design, polymorphism etc).

After 60 minutes we collected all exams, made them anonymous, and redistributed them for marking (no student marked his or her own exam). The marked exams were then submitted for a final control.

- A *practical examination* (in-lab, 3 hours) at the end of the course.

  Conventional theoretical exams are not sufficiently powerful at testing the practical abilities of a student. This is usually the task of the assignments. We wanted a more thorough test of each student's ability to independently solve practical problems. Any aids, except peer-to-peer communication, were allowed. Even searching the Internet was allowed, but not advisable, since time was quite limited.

  This set-up gave us the opportunity to see whether both groups had reached similar levels of practical problem solving skills.

- Completion of *assignments*.

  Students were given three assignments with increasing complexity. Assignment 1 and 2 involved the modification/ completion of given applications. For Assignment 3 students had to develop an application/applet from scratch and were allowed to choose one out of four tasks.

- Monitoring of the *CRC card sessions*.

  During the sessions we observed the working groups and made notes on their performance.

- A *course evaluation* at the end of the course.

Attendance was recorded for all in-lab and group sessions, but not for the lectures. For students who were unable to participate in both parts of the examination (theory and practice), a traditional written re-exam was offered late August, since taking only one of the tests would not do.

RESULTS

The overall course results for the redesigned course were promising. The drop out rate was lower (37,6%) and the pass/fail ratio better (47/16) than usual for a summer course.

The final results for our study groups, after the re-exam, are summarised in table 3.

TABLE 3
OVERALL RESULTS BY GROUP

	Drop out	Fail	Pass	Good	Distinction
G1	3	5	1	3	0
G2	2	7	5	0	0

The *short quiz* was taken by 8 (out of 12) members of group G1 and 12 (out of 14) members of group G2. The overall performance of G1 was substantially better than that of G2. The results are summarised in table 4 below with specific noteworthy differences highlighted.

We had only two questions where G2 slightly outperformed G1 (no. 2 and 11) both dealing with low-level

details of Java. These differences were expected as well as G1 showing a much better understanding of the concepts class and object (no. 5). Somewhat unexpected were the significant differences in questions 4 and 8, since both topics had not yet been covered in any of the group sessions. The difference in question 9 was less marked as expected. However the results indicate that G1 showed a better understanding of general object-oriented concepts, whereas G2 better mastered syntactical details.

TABLE 4
Quiz Results by Question in % of Possible Points

No	Topic	G1	G2
1	Language syntax vs. semantics	73.3	63.9
2	Valid identifiers	60.0	63.9
3	Constants vs. variables	76.7	72.2
4	Priorities	100	69.4
5	Class vs. object	100	77.8
6	Formal vs. actual parameters	20.0	13.9
7	Purpose of constants	76.7	72.2
8	Code understanding (loops)	40.0	19.4
9	Purpose and usage of private	73.3	63.9
10	Code understanding (constructors)	63.3	55.6
11	Code understanding (expressions/types)	36.7	41.7
	TOTAL	65.5	55.8

Even the *theoretical examination* showed significant differences between both groups. Participation was the same as for the quiz.

The overall performance of G1 was still substantially better than that of G2. G1 showed far better conceptual knowledge (questions 4-7) whereas G2 generally showed a better understanding of actual code (questions 3, 10, and 12), except array manipulation (question 9). Surprisingly there were no marked differences in the design question (no. 8). Please note the significant difference in results for the two questions on polymorphism (no. 7 and 12). Performance of G2 was the same for both questions, whereas G1 had a marked difference between concept level knowledge (question 7) and code level (question 12). The results are summarised in table 5.

TABLE 5
Results of the Theory Exam by Question in % of Possible Points

No	Topic	G1	G2
1	Kinds of programming errors	81.3	72.9
2	Meaning of 'strong typing'	12.5	12.5
3	Find errors in piece of code	37.5	58.3
4	Method signatures	75.0	45.8
5	Overloading vs. overriding	68.8	41.7
6	Usage of modifiers	56.3	41.7
7	Inheritance (polymorphic variables)	100	58.3
8	Design exercise (calculator)	39.6	33.3
9	Code understanding (array manipulation)	53.1	40.6
10	Local variables and shadowing	0	8.3
11	Purpose and usage of super (...)	81.3	72.9
12	Code understanding (polymorphism)	52.1	58.3
13	Abstract classes vs. interfaces	50.0	45.8
	TOTAL	56.7	47.3

The *practical examination* was taken by 5 students of G1 only, including the two best performers from the theory exam. The exam comprised six exercises. Exercises 1 and 2 were obligatory. Two more could be chosen from exercises 3-6. It is therefore difficult to draw any conclusions without examining the solutions more closely. The results are summarised in table 6 for completeness.

TABLE 6
Results of the Practice Exam by Question in % of Possible Points

No	Topic	credits	G1 #	G1 %	G2 #	G2 %
1	Implement method (call provided)	6	5	48.3	7	40.5
2	Design for a stop watch	6	5	66.7	9	70.4
3	Implement applet (coloured strings)	4	3	50.0	9	79.2
4	Implement method (call provided)	4	2	56.3	4	46.9
5	Modify applet (add method & button)	6	2	91.7	5	55.0
6	Modify PacMan (add classes)	8	1	87.5	0	—
	AVG TOTAL CREDITS			12.6		10.75

#: Number of students attempting to solve exercise.

The recording of *assignment completion* did not reveal any significant differences (see table 7). To draw any conclusions the solutions need to be examined more closely.

TABLE 7
Assignment Results in % By Group (Excluding Drop Outs)

	assignment 1 OK	assignment 1 on time	assignment 2 OK	assignment 2 on time	assignment 3 OK	assignment 3 on time
G1	100	100	100	100	77.8	88.9
G2	100	100	100	100	91.7	91.7

Monitoring of the *CRC card sessions* did not reveal any differences between the groups. However, we noticed that experienced programmers often bogged down their groups in irrelevant implementation details. Two such groups never got far enough to actually role-play a scenario. Groups with inexperienced programmers only were very successful and came up with sensible designs, but had problems transfering their results into actual Java designs/ code later on.

STUDENT FEEDBACK

After the course we administered a questionnaire to get some feedback from the students. Unfortunately only 27 students out of 63 active ones (42.9%) returned the form. Of these students 8 (4+4) were from our study groups.

We asked the students how they felt about BlueJ, the CRC method, the course's structure and examinations, and whether they would recommend the course to fellow students. In addition we asked for suggestions for the next summer course.

All students using BlueJ (12) were generally positive about it ("A good way to visualise object interconnections." [member of G1]). However five students mentioned usability problems, mainly concerning the integrated editor.

Even the CRC method was very well received by the students. Only one student out of 19 wrote a negative comment and three more were neither positive nor negative.

"... an opportunity to control one's ideas early." [G1]

"Very good exercise before assignment 3." [G2]

"This exercise really helped me to understand! It was nice to see that there are several solutions to a problem."

Students also felt very positive about course design and examination forms. There was only one negative student response out of 27. However several students complained that too little time had been allocated for the theory and practice exams.

All 27 students would recommend the course to their fellow students.

We received many suggestions for improvements. The following ones were mentioned more than once. Five students suggested lowering the level of the course, especially in the beginning. On the other hand four students suggested more difficult tasks for assignments 1 and 2 and two students would like to see more advanced Java topics. Two students suggested more group work.

DISCUSSION

The data sets for this study have been small. The results are therefore quite sensitive for noise. Several students in our study groups for example did not participate in one or more of the examinations. The practical examination was taken by 14 (5+9) out of 26 students in our study groups. However we did not lose all of the best, worst, or average performers in either group. The student changing study group from G1 to G2 performed quite well. Had he stayed in G1 the differences between the groups would have been even more marked.

Female participation was quite high in our study groups. It can be noted that of the 8 female students in G1 3 dropped out and only one of the remaining five passed the course. On the other hand did 3 of 4 male students pass the course. The situation in G2 was not much better. On the other hand our observations show that female students performed very well in the CRC sessions and enjoyed them very much. The gender differences might be explained by the background of our subjects. Most men have a background in science/technology programs, whereas many women have a background in Information Systems. Unfortunately we did not collect sufficient data from our subjects about this kind of background.

Assignments were only graded as either passed or failed. We did not analyse in detail how students solved them. However, doing so would have resulted in a much better insight into their actual programming abilities.

FUTURE WORK

This study gave us confidence in further pursuing the goals stated in the beginning of this paper. For spring 2002 we have adopted the new approach (G1) for a Java course for media engineers. However, to fully reach our goals more work is necessary in some areas.

The order of lectures must be "optimised" by putting more BlueJ/CRC related topics into the first lectures.

There still is some "magic" in some of our examples. We will therefore work on further examples that can be used during the in-lab sessions and CRC role-plays.

Since there still are no textbooks supporting our approach we will also develop further handouts and step-by-step instructions for our hands-on lab sessions.

For the next summer course we have planned a case study to evaluate student results against their backgrounds with respect to gender, mathematics, and programming.

CONCLUSIONS

We presented a list of goals for the design of an object-centric holistic approach to the teaching of object-oriented programming with Java. To accomplish the goals proper tools and examples are imperative. Our usage of the BlueJ programming environment and the CRC card technique together with hands-on exercises employing carefully designed examples, seems to stimulate the understanding of object oriented concepts.

The results of our case study indicate faster and better understanding for object-oriented concepts through delaying the introduction to complete applications and I/O. Being able to explore classes and methods without worrying about the context in which they appear, supports understanding without the confusing syntactical details.

REFERENCES

[1] Becker, B. W., "Pedagogies for CS1: A Survey of Java Textbooks", manuscript, http://www.math.uwaterloo.ca/~bwbecker/papers/javaPedagogies.pdf, last visited 2002-03-01.

[2] Bellin, D. and Simeone, S. S., The CRC Card Book, Addison-Wesley, 1999.

[3] BlueJ Home Page, http://www.bluej.org, last visited 2002-03-01.

[4] Guzdial, M., "Centralized Mindset: A Student Problem with Object-Oriented Programming", Proceedings of the 26th Conference on Computer Science Education, USA, Mar 1995, pp. 182-185.

[5] "'Hello, World' Gets Mixed Greetings", Forum of the Communications of the ACM, Vol. 45, No. 2, Feb 2002, pp. 11-15.

[6] Kölling, M. and Rosenberg, J., "Guidelines for Teaching Object Orientation with Java", Proceedings ITiCSE, Canterbury, UK, Jun 2001, pp. 33-36.

[7] Lewis, J. and Loftus, W., Java Software Solutions, 2nd edition, addison-Wesley, 2001.

[8] Morelli, R., Java, Java, Java—Object-Oriented Problem Solving, Prentice Hall, 2000.

[9] Thimbleby, H. "A Critique of Java", Software—Practice and Experience, Vol. 29, No. 5, 1999, pp. 457-478.