Robots: Learning to Program with Java
A Publication Prospectus

Byron Weber Becker
School of Computer Science
University of Waterloo
Waterloo, Ontario N2L 3G1
Canada

bwbecker@uwaterloo.ca
http://www.cs.uwaterloo.ca/~bwbecker/
31-January-2004

1 Summary

Robots: Learning to Program with Java is an introductory programming textbook nearing completion. This prospectus explains its approach to teaching object-oriented programming, outlines the benefits of this approach, compares *Robots* to existing textbooks, and discusses its special features and development schedule.

The major parts of this prospectus are:

1 Summary 1
2 Description 1
 2.1 Approach 2
 2.2 Advantages of Using Robots 3
 2.3 Object-Oriented Pedagogy 4
 2.4 Table of Contents 5
3 Ancillaries and Special Features 7
4 The Market and the Competition 8
 4.1 The Market 8
 4.2 Comparison with Other Textbooks 9
  4.2.1 Objects Early: Use Author's Classes, then Write (1) 9
  4.2.2 Objects Early: Use Java's Classes, then Write (2) 11
  4.2.3 Objects Early: Write and Use (3) 12
  4.2.4 Objects Early: Write, then Use (4) 12
  4.2.5 Objects Superficially Early (5) 13
  4.2.6 Objects Late (6) 13
 4.3 Recommendations from Others 13
5 Development Schedule 14
6 Potential Reviewers 14
7 Vita 15

2 Description

As often happens, this book was born because the author was unhappy with the alternatives. When I was first asked to develop a Java version of our introductory programming course for 1,000 students per year, I naturally collected all the relevant Java textbooks I could find. They all left me with a vague sense of

uneasiness. Yes, the programming language had changed from Pascal to Java, but the approach to the material had not. A second shift was necessary: a shift in pedagogy.

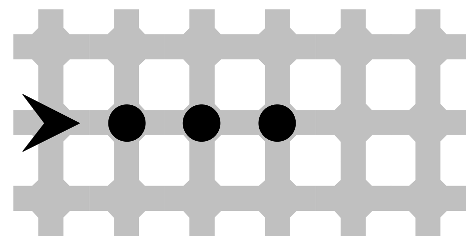
The first term of my course did not go well. Halfway through the second term I discovered a small book, *Karel++: A Gentle Introduction to the Art of Object-Oriented Programming* (Wiley, 1997). It was an "*Aha!*" experience for me. The pedagogy of this book felt right to me and I knew its metaphor of programming robots would appeal to my students. It had an obvious appeal for visual learners and I could imagine having lots of fun acting out programs with students. Unfortunately, *Karel++* is a C++ textbook, not Java. Furthermore, at only 175 pages and lacking many language-specific details, it forms the first several weeks of an introductory course. After that, a different textbook is required; a textbook that did not exist.

Discussions with the publisher of *Karel++* led to permission to translate it to Java for use at the University of Waterloo. After experiencing the joys of teaching with the approach – and the difficulties of changing to an unrelated text after a few weeks – I began to write the textbook I really wanted. *Robots* combines the wonderful pedagogy of *Karel++* with the full and complete treatment required by an introductory object-oriented programming textbook.

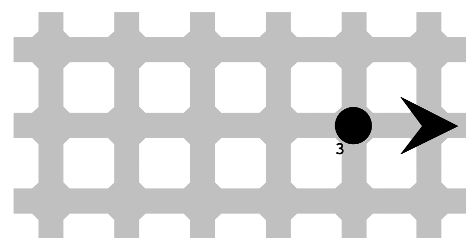
2.1 Approach

Robots uses the metaphor of programming a robot to teach programming in general. Robots can move, turn, pick things up, carry things, and put things down again. Students have a mental image of robots and can easily direct them to perform a task such as picking up three things in a row and putting them in a pile. This task can be clarified with a pair of diagrams, below. The first shows how the task begins: with the robot (an arrowhead) and three things in front of it. The second diagram shows how the task should end. A student can easily "program" another student or the instructor, named "Hal," to complete this task by saying:

Hal, move
Hal, pick up a thing
Hal, move
Hal, pick up a thing
Hal, move
Hal, pick up a thing
Hal, move
Hal, put down a thing
Hal, put down a thing
Hal, put down a thing
Hal, move



Initial Situation



Final Situation

After students verbally direct "Hal", it is easy to introduce a simple program that does the same thing where `hal` is the name of a robot object.

```
hal.move();
hal.pickThing();
hal.move();
hal.pickThing();
hal.move();
hal.pickThing();
hal.move();
hal.putThing();
hal.putThing();
hal.putThing();
hal.move();
```

Obviously, there are additional details to cover before this Java fragment can be executed as a complete program. However, these details can be taught as a pattern, leaving the focus on using robot objects to accomplish tasks.

Several sample programs are available at http://www.learningwithrobots.com under the "Robot Examples" link. One example is an introductory program, such as just discussed. Another instructs a robot to make a bar graph, an example we use in lecture to illustrate stepwise refinement. A third example shows several robots working together to make a bar graph while a fourth uses threads so that all the graphing robots move simultaneously. Finally, an advanced example sets up robots as checker pieces, allowing two people to play a game.

Lest students believe that what they learn applies only to robot objects, the end of each chapter applies the same concepts in another context, often using graphics. In chapter 1, for example, students learn how to create robot objects and call their methods. They also learn how to create a `JFrame` object and calls its methods.

2.2 Advantages of Using Robots

There are many significant advantages to using the robot metaphor to learn programming. Four of the most important are outlined here.

First, the visual qualities of robots are a wonderful advantage. They benefit instructors and students in a number of ways.

- The visual qualities of robots make it easy to specify a problem to solve. Many problems can be specified using a picture of the initial situation and another of the final situation, plus a few lines of text. We find that students have fewer questions about homework problems specified this way.
- Robot programs provide visual feedback on the correctness of an algorithm. If the student's program doesn't result in the same image as the problem statement, there must be a bug.
- When a student's program does not end with the same image as the problem statement, students can often see where their program goes wrong simply by watching the animation. Because the human brain is highly optimized to process visual input, this is faster and easier than, say, scanning a list of numbers from a console program.
- Teaching traditional input and output can be delayed because robots provide output visually. This is particularly attractive in Java where input and output are cumbersome, at best.

Second, object-oriented programs are easier to write when programmers can imagine what they would do if they were the objects in the program. Robot objects make this easy. Because moving, turning, picking things up and putting them down again are things that we do every day, it is easy for students to give directions to one another or to an actual robot object. Either way, students are learning important programming skills by identifying with the objects they use.

Third, robots are fun! I have never had so much fun with a classroom of students as the day we worked with a "paranoid" robot that "looked" to the right and to the left before it moved forward. People who acted it out adopted a hunched, uptight look with shifty eyes that generated much laughter among the students. Later in the same period we turned this into a paranoid thief that went up the aisle swiping small objects from student desks, all the while looking both ways before it would move. It was fun, but it also taught students about inheritance, one of the three hallmarks of object-oriented programming.

Finally, I believe that the largest benefit of using robots is that they lend themselves to a superior pedagogy for teaching object-oriented programming. This ultimate benefit is more fully explained in the following sections.

2.3 Object-Oriented Pedagogy

Every textbook author faces a multitude of decisions. In tackling an object-oriented programming textbook, there are two decisions that fundamentally affect the text and its pedagogy: when to introduce objects and how to introduce objects.

Early Java textbook authors usually decided to place objects late in the textbook, making the books feel like rewritten Pascal textbooks. The conventional wisdom was that objects is an "advanced" topic that is too difficult for students to learn early.

It feels wrong, however, to leave the major idea in object-oriented programming to the end of the course. If possible, the major idea should be first. This maximizes the amount of time students can practice and build on the idea. It also prevents a difficult paradigm shift later in the course from a procedural style (like Pascal) to an object-oriented style.

Most current textbooks introduce objects early in the book, at least superficially. *Robots* follows this trend. It goes further, however, by meeting a list of six criteria for example programs designed to maximize student understanding of how objects work. I enumerated these in a letter published in the *Communications of the ACM*[1] (quoting):

1. Early examples ought to use objects, the central feature of the paradigm.
2. Objects should be explicitly instantiated (unlike `Strings` and `System.out`). Understanding how objects are created is a vital part of learning to think with objects.
3. Objects should have their methods invoked, otherwise students view them simply as data containers or abstract pieces of syntax.
4. Each object should have easily discernable state and behavior. If not, the two core aspects of an object will remain a mystery to students.
5. Examples should contain two or more objects from the same class to drive home that each object has its own state but shares behavior with other members of its class.
6. Static methods, other than "main", should be avoided because they don't affect the state or behavior of individual objects, thus clouding two core concepts.

Robots meets all of these important characteristics. Very few, if any, other textbooks manage this.

The second major decision object-oriented textbook authors must make is how to introduce objects. Recall that the concepts of object and class are intimately related. Each kind of object in a student's program is created from a class that a programmer writes to define the objects' characteristics. Given that students need to master both using objects and writing the classes that define them, a crucial question is how these topics should be ordered. There are three possibilities for *writing* classes and *using* the resulting objects:

- *Write and Use*: In this approach students are asked to master the basics of writing a class at the same time they are learning how to use objects. Horstmann, for example, introduces classes and objects by describing how to use a bank account object in 2 pages. He then delves into the details of writing the class to define it. This requires introducing students to the distinction between class and object, declaring objects, object instantiation, invoking methods, the structure of a class, defining methods, declaring formal parameters and using actual parameters, return values, and instance variables. This presents an incredible cognitive load for students. Horstmann chose a wonderful example to convey all these concepts, but it is still difficult to understand all the concepts, even at an introductory level.

[1] *Communications of the ACM*, Feb. 2002, Vol 45, No. 2, page 11.

- *Write, then Use*: When actually writing a program, programmers first write the required classes and then use the objects they define. I am aware of only one textbook, by Nino & Hosch, that has chosen to follow the same ordering. They include a light treatment on the idea of an object, but then delve into the details of writing classes with very few examples of how the objects they define would be used. This lessens the cognitive load on the students by focusing on just one of the two aspects, but leaves students wondering how the classes they are writing fit into the larger picture. Much of the instruction on writing classes is lost because students don't have practical experience in using the resulting objects.
- *Use, then Write*: A third possibility is to use objects from existing classes and then learn how to write your own classes. This possibility exists because the student does not necessarily have to write a class to use the objects it defines. *Robots* uses this approach. Students make extensive use of robot objects, learning how to declare objects, instantiate objects, and invoke their methods. All the details of writing their own classes come later, after they are comfortable with using objects.

Obviously, I believe that *Use, then Write* has more merit than the other two possibilities. It is not a new idea. Rich Pattis[2], the originator of teaching with robots, enumerated the following advantages of *Use, then Write* in 1993 when students learning Pascal were often asked to write subprograms at the same time they were learning to use them. With a few substitutions, the arguments are still valid:

- The experience of using subprograms (objects) helps students understand what they are and why they should be used.
- Students learn to read and use libraries (of classes), a fundamental skill.
- Students become "documentation consumers, not producers" which is more apt to teach the need for quality documentation.
- Students can design and implement more sophisticated and satisfying programs than they could on their own.

A third decision facing textbook authors who choose *Use, then Write* is choosing the objects students should use first. There are two basic alternatives: objects defined by classes that come with Java, and objects defined by classes provided by the author. *Robots* uses the second alternative. The robot objects have been carefully crafted for teaching beginning programmers. The resulting programs are visual and animated. The classes are complex enough to be interesting, yet not burdened with features not of interest to students. These are advantages that cannot be found when using classes provided by the Java library.

In summary, *Robots* is a powerful, yet gentle objects-early pedagogy. It is also visual and fun for both instructors and students. Most significant, however, is that students start using objects immediately. Only after learning how to use objects they are introduced to the complexities of defining their own objects by writing classes.

2.4 Table of Contents

Eight of the thirteen chapters in *Robots* are nearly complete. All chapters are summarized here; please see the manuscript for complete details.

Chapter 1: Programming with Objects

Writing a `main` method that instantiates one or more objects and invokes their methods to accomplish a task. Students understand – both from reading and from the programs they write – that each object maintains its own state but shares behaviors with other objects in its class.

Chapter 2: Extending Classes with Services

[2] Pattis, Richard E. "The 'Procedures Early' Approach in CS1: A Heresy" *SIGCSE Bulletin* 1 (1993), p. 122-126.

Already in Chapter 1 students will ask, "Can a robot turn right?" "Can it move five times?" Primed for more sophisticated robots that can perform services helpful for their immediate task, students find it easy to extend the class with new methods. This accomplishes two goals: introducing procedural abstraction and introducing inheritance.

Chapter 3: Developing and Reusing Methods

In chapter 2 the algorithms are very simple. In chapter 3, more substantial algorithms are tackled that make pseudocode and stepwise refinement invaluable. Students also see that by overriding key methods in a superclass, they can easily create classes that do different, but related tasks.

Chapter 4: Variables

Classes contain two important kinds of things: methods (discussed in chapters 2 and 3) and instance variables. This chapter looks inside the `Robot` class to see how instance variables work in a familiar context. Other kinds of variables (temporary or local variables, parameter variables, and constants) are also examined in both robot and non-robot examples.

Chapter 5: Making Decisions

Robots introduces `if` and `while` statements at almost the same time. Students often confuse the purpose of these two statements. By treating them side-by-side, their similarities and differences can be highlighted. Chapter 5 also includes a discussion of simple Boolean expressions, nesting control structures and writing predicates to use in `if` and `while` statements.

Chapter 6: More Decision-Making

This chapter expands on the previous one. It includes a four-step process for writing `while` loops, variations of the `if` statement such as `if-else`, cascaded `if-else`, and rewriting `if` statements. It also includes looping variations and compound Boolean expressions.

Chapter 7: More on Variables and Methods

Chapter 7 expands on Chapter 4 with more about primitive types (Booleans, characters, etc), more complex expressions, the `static` keyword, and methods common to all classes such as `toString` and `equals`. Most of this chapter remains to be written.

Chapter 8: Collaborative Classes

Collaborative Classes emphasizes how two or more classes can work together. This chapter is still unwritten.

Chapter 9: Input and Output

Reading information from the console as well as from files is the focus of this chapter. It also touches on file formats, writing to files, writing a simple command interpreter, and creating a personal library of useful classes. It finishes with an introduction to the Java I/O package.

Chapter 10: Arrays and Collections

This chapter begins by using arrays. Assuming an array exists, what can we do with it? Once the usefulness has been demonstrated, we look at the nitty-gritty details of creating arrays and more advanced usage such as partially-filled arrays. The chapter ends by examining the Java collections library. This part is still unwritten; it is my intention to make use of Java 1.5 generics.

Chapter 11: Building Quality Software

What defines quality software? What kind of disciplines are useful to write it? This chapter is still unwritten.

Chapter 12: Polymorphism

In lectures we use a wonderful example based on robots to illustrate polymorphism. I intend to do something similar in the text before branching out into inheritance hierarchies. This chapter is not yet written.

Chapter 13: Graphical User Interfaces

More than just using Swing, this chapter will introduce the Observer pattern and separate the model's implementation from the view. The chapter is not yet written. The material, however, has been well-developed and tested in lectures.

Appendices

A full set of appendices includes documentation for key parts of the provided software, a glossary, and so on. It also includes a novel section that gives an overview of many important concepts. It may be used either as a preview for students who are already familiar with programming or as a review before the midterm or final exam.

3 Ancillaries and Special Features

Robots comes with significant ancillaries and an impressive list of features. The primary ancillary, of course, is the library containing the robot classes and classes for simplified input and output. These libraries work with any Java development environment (JDK 1.3 and above) and permit students to write and run robot programs. Because a regular development environment is used, students do not experience a transition in technology from writing robot programs to any other kind of program.

Another significant ancillary is two complete sets of lecture notes. These notes are for two groups of students: those who have some programming experience (and can therefore move much faster), and those who have not. At the University of Waterloo, this represents one course (CS133) for those with prior programming experience, and a sequence of two courses (CS131 and CS132) for those without such experience. The two course sequence covers almost exactly the same material as the single course, but at a slower pace.

These lecture notes are living documents, always being updated with our most recent ideas for how to teach more effectively. As the *Robots* textbook is completed, the lecture notes will integrate with it increasingly well. As it stands now, the slower CS131/132 sequence matches the text fairly well; the CS133 notes are still evolving to use the text more thoroughly.

All three courses use an innovative, active learning approach to teaching. A weekly cycle begins with students reading a selection from the textbook. Students and instructors then meet twice. The first is a lecture in which the instructor uses examples to show how he or she would solve a problem using techniques the students have read about. In the second meeting, students are sometimes introduced to additional techniques, but more often they are given similar examples and asked to design and/or implement key parts. While students do this, the instructor monitors the individuals in the class, helping where needed. As instructors, we enjoy the mix of being the "sage on the stage" *and* the "guide by the side." At the University of Waterloo, lectures are in groups of 90 while the second meeting, the "Practicum", occurs in groups of 45.

Samples of the lecture notes and slides supporting them are included with this package. The instructor notes include thumbnail images of the slides shown to students. At Waterloo, we provide copies of many (but not all) of the slides to students.

The features of the textbook itself include the following:

- A *Use, then Write* pedagogy in which students learn about classes and objects by using them before being asked to write their own. This approach has been classroom tested at the University of Waterloo for more than five years with about 1,000 students per year.
- A supporting library, `becker.jar`, containing the classes needed to write, run and animate robot programs. It also includes:

    - classes for simplified input and output. The `TextInput` class includes innovative queries such as `intIsAvailable()` so students can easily write robust code that detects and recovers from user input errors.
    - classes implementing graphical user interfaces for a growing collection of programming assignments. Students write classes that are combined with the user interface to create a complete and satisfying program.
- Margin notes in three different flavors:
    - *Key Idea* margin notes highlight the introduction of a key idea. They provide a quick summary for review and an easy way to scan for important concepts.
    - *Looking Ahead* margin notes link material with what is to come. These notes provide an easy and effective way to tell the students that this is not yet the whole story and directing interested students to deeper explanations.
    - *Looking Back* margin notes do the reverse – they remind students where they first read about a concept in case they need a quick review before plunging ahead into new material that uses the concept.
- An appendix, "Lay of the Land," is designed specifically for students with previous programming experience. It gives an overview of much of the material and allows them to match their current knowledge with what they will learn. For example, this material will allow many students to apply their previous knowledge of repetition and selection long before they read about all the details in chapters 5 and 6. This material may be either used or ignored at the instructor's discretion. An alternate use is for a quick review before an exam.
- A glossary defines all the terms introduced in the textbook. Terms are italicized when first introduced. The index identifies terms contained in the glossary with an italicized entry for the glossary and a bold entry for the first use in the text.
- Many illustrations, facilitated by the visual nature of robots.
- Class diagrams using the Unified Modeling Language (UML) are used to communicate program designs.
- Many complete programs. Complete source code is available from the book's web site at http://www.learningwithrobots.com.
- Chapters that each contain:
    - an overview and chapter objectives.
    - discussions of common programming patterns associated with the chapter's topics; an icon appears in the margin beside each usage of a pattern within the chapter.
    - a concept map showing how the concepts introduced in the chapter are related to each other; see Chapters 1 and 5 for examples.
    - extensive problem set that give students an opportunity to practice what they learn.

4 The Market and the Competition

4.1 The Market

The target market for *Robots* is introductory programming courses for either majors or non-majors at the college and university level. Some high school teachers have also used it for advanced courses that introduce Java.

Robots should be the only resource students require except for a computer and software that is freely available on the web or could be provided on a CD-ROM with the textbook.

This is admittedly a crowded market and *Robots* is a late entry. Nevertheless, I think that *Robots* will be successful because of the advantages cited earlier (a solid objects-first pedagogy, visual representation, fun). Furthermore, the introductory programming market is not monolithic. We are still finding our way pedagogically and the market reflects this. When pedagogy is taken into account, *Robots* has only five competitors. This is elaborated in the next section.

4.2 Comparison with Other Textbooks

Earlier in this proposal is a discussion of three important decisions confronting Java textbook authors:

- Should objects – the fundamental paradigm – be presented early or late?
- How should writing classes and using the objects they specify be ordered: *Write and Use*; *Write, then Use*; or, *Use, then Write*?
- If students use objects before writing their own classes, should those objects be from the Java library or objects developed specifically for teaching?

These three decisions can be diagramed as a tree to classify existing textbooks (see Figure 1). This tree helps identify the true competition for *Robots*: those books with a "*Use author's classes, then Write*" objects-early approach. The comparison of *Robots* to these five books is presented first. For completeness, the remaining five categories of books are also discussed.

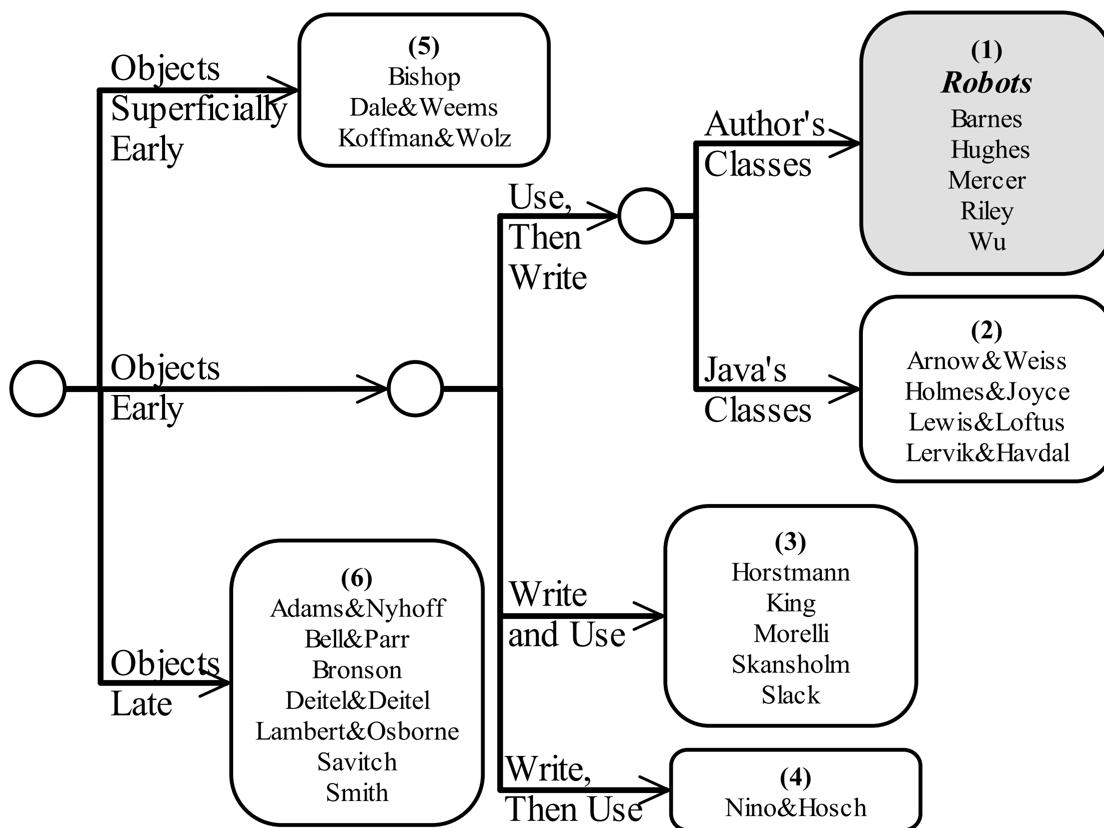


Figure 1: A classification of some existing introductory Java textbooks.

4.2.1 Objects Early: Use Author's Classes, then Write (1)

This is the category to which *Robots* belongs. The other five books in this category represent the most natural competition to *Robots*, assuming instructors will gravitate towards books with similar pedagogies. The competition is:

- *Object-Oriented Programming with Java: An Introduction* by David J. Barnes (Prentice Hall, 2000)
- *Fundamentals of Computer Science using Java* by David Hughes (Jones & Bartlett, 2002)
- *Computing Fundamentals with Java* by Rick Mercer (Franklin, Beedle & Associates, 2002)
- *The Object of Java* by David D. Riley (Addison Wesley, 2002)
- *An Introduction to Object-Oriented Programming with Java* by C. Thomas Wu (McGraw Hill, 2001)

As little as two years ago, this category had only two entries other than *Robots*: the books by Wu and Barnes. Within the last two years the majority of the new introductory Java textbooks I have received fit here. The idea of teaching Java by having students use classes specifically crafted for teaching is catching on – and textbooks are finally becoming available to support this trend.

All six of these books come with classes for students to use as they learn about objects, of course. *Robots*, however, is different from them in a number of ways.

1. Classes like ships, pencils, grids, and bank accounts just aren't as interesting as robots. They aren't as fun to role play, the students can't direct their activities as intuitively, and students can't identify with them as easily in program design activities. The basic metaphor of programming a robot is tremendously attractive as an instructional aide and should not be underestimated.
2. *Robots* often uses two or more robot objects in the same program. This demonstrates that objects from the same class share the same behaviors but have different state. This also shows that each object maintains its own state, independent of all the other objects in the system. Barnes, Riley and Mercer have a few examples that do this, but students may not even notice them illustrating this basic principle. Such examples don't make sense with Wu's classes and cannot be done with Hughes' classes.
3. The set of classes provided with *Robots* is rich enough to illustrate many computing concepts: using objects, extending classes with new methods, repetition and selection, polymorphism, and so on. One example can build on another. Students aren't required to learn a new context, with the overhead that entails, as often. In the other five books, the classes provided are much more limited, used for a little while, and then discarded. But *Robots* does not focus only on the robot classes. Students require more than one context to adequately understand how classes and objects work. This is provided by a second track, focusing on graphics. Later in the book, after students have a solid footing with object-oriented programming, the robot objects are left behind in favor of a wide variety of classes.
4. The robot classes are clearly a system of interacting classes. Students learn early that a real program requires different kinds of classes, all working together to solve the problem. *Robots* has a little extra complexity to get started, but this complexity fits a well-defined pattern and is easily managed. The other five books often have multiple classes in their examples, but they take pains to hide this fact from the students. This is the root cause of Hughes' inability to use two or more objects from the same class, cited above.
5. Along with the three newest books, *Robots* includes design patterns, a concept that has swept the industry. *Robots* identifies many elementary patterns throughout the text with an icon in the margin where they are used and a section in the text explaining the pattern in more detail. The other three textbooks, however, don't follow through with the idea as thoroughly as *Robots*. Patterns are sporadic, or only late and complex, or are not well-identified.
6. *Robots* has the name and reputation of the University of Waterloo behind it. In Canada, UW is recognized as *the* place to go for undergraduate computer science education. Many high schools and other colleges and universities watch Waterloo's computer science curriculum very closely. At ACM's International Collegiate Programming Contest finals, Waterloo teams have placed in the top 10 for ten of the last eleven years. In years past, UW provided instructional tools such as WatFor, WatFiv, and WatCom Pascal (programming languages) and MacJanet (a networking system designed

for educational use). Many educators all over the world still remember the significant educational contribution that UW made via these tools. Along with the name and reputation, Waterloo brings a sizeable market of more than 1,200 students per year who use *Robots*.

Finally, the competition in this segment includes a Java version of *Karel++*, the book that got me started on this project. It is currently available only on-line at

http://csis.pace.edu/~bergin/KarelJava2ed/Karel++JavaEdition.html

Karel J. Robot differs from *Robots* in that it is not a complete textbook. It must be supplemented or followed by a another Java textbook to learn about core topics such as variables, arrays, file input and output, and so on. This is unsatisfactory because there are no good follow-on textbooks that account for what students have already learned with *Karel J. Robot*.

4.2.2 Objects Early: Use Java's Classes, then Write (2)

The four books in this category represent another way to implement a *Use, then Write* approach to teaching objects early – by using Java's classes instead of author-supplied classes. These books are:

- *Introduction to Programming Using Java: An Object-Oriented Approach* by David Arnow and Gerald Weiss (Addison Wesley, 2000)
- *Object-Oriented Programming with Java*, by Barry J. Holmes and Daniel T. Joyce (Jones and Bartlett, 2001)
- *Java Software Solutions*, by John Lewis and William Loftus (Addison-Wesley, 2000)
- *Java the UML Way: Integrating Object-Oriented Design and Programming*, by Else Lervik and Vegard B. Havdal (Wiley, 2002)

The book by Arnow and Weiss is the best example of this approach. In Chapters 2 and 3, they clearly focus on using objects created from classes in the Java library. Their choice is to focus on objects that read data from and write data to files on the disk drive, with some secondary focus on Strings. There is much to learn about using objects in general by using these particular objects, but they are difficult to use. In fact, many authors provide their own, easier to use, objects for reading and writing from a file simply to avoid Java's inherent complexity in this area. When Arnow and Weiss do introduce students to writing classes, the first examples are not very interesting nor is it obvious that someone would use them in a real system.

In spite of these short-comings, the textbook is well thought-out. I like the supplement introducing graphics at the end of each chapter, and have borrowed and adapted the idea for use with *Robots*. This text is also one of the few that deals with classes that interact with each other in non-trivial ways. Their truck and toll-booth example is exemplary in this area.

The books by Lewis & Loftus and Holmes & Joyce both avoid the difficulties of Java input and output by using String objects and an object provided automatically in every Java program, `System.out`. The problem with this approach is that both Strings and `System.out` enjoy special support within the Java language, obscuring important properties of objects. Furthermore, it isn't obvious what state the `System.out` object contains and any state maintained by a String cannot be modified, again hiding important properties of objects. These books do introduce a few other classes from the Java library that do not have some of these problems, but the result is less than satisfactory.

Lervik and Havdal introduce students to using objects with a class modeling a car. Unfortunately, they don't provide the class and so students can't actually write the program they show in the book. The first objects that students can actually use for themselves are from the Java library – Strings (again) and a class for generating random numbers.

In summary, the books in this category either fall into the trap of relying heavily on objects that are somehow special (Strings and `System.out`) or they fall into the trap of using objects that are difficult to use. Neither is a particularly good way to introduce students to object-oriented programming.

4.2.3 Objects Early: Write and Use (3)

Five textbooks using the *Write and Use* flavor of the objects early approach are

- *Computing Concepts with Java2 Essentials*, by Cay Horstmann[3] (Wiley, 2000)
- *Java Programming: From the Beginning*, by K.N. King (Norton, 2000)
- *Java, Java, Java: Object-Oriented Problem Solving*, by Ralph Morelli (Prentice Hall, 2000)
- *Java: From the Beginning*, by Jan Skansholm (Addison-Wesley, 2000)
- *Programming and Problem Solving with Java*, by James M. Slack (Brooks/Cole, 2000)

These five books all attempt to truly teach objects early, but demand that students master a huge number of details at one time. Of the five, I think *Computing Concepts with Java2 Essentials* by Cay Horstmann is one of the best. His first example, a class modeling bank accounts, is very simple and leverages our intuition from real bank accounts. However, the first program that uses this class occurs only after 120 pages of details and even then is quite boring: crediting interest to a bank account. There are earlier programs, but they are simply illustrating language details such as working with integer variables.

Furthermore, the number of concepts that students must grasp for even this simple class to make sense is staggering. Students must learn about the structure of a class, methods, formal parameters, instance variables, return statements, instantiating an object, invoking methods, and passing actual parameters, all in the space of just a few pages. This is simply too much, too fast.

The other four books in this category suffer from the same basic problem, with each adding its own idiosyncrasies. For example, Skansholm uses many examples of defining classes, but no programs that actually show how to use the resulting objects. Morelli's first class, a "cyber-pet" that eats and sleeps, suffers from the same criticisms as Horstmann, plus the fact that it makes extensive use of Boolean variables. Students always have a harder time understanding Boolean variables than much more familiar integers. King's first example is also a bank account and has the same criticisms as Horstmann.

The textbook by Slack is nearly in the *Use, then Write* category. The entire second chapter makes extensive use of an object implementing turtle graphics, an idea that has been around for at least 25 years. This would seem to be *Use, then Write* except that the author does not even point out that the turtle is an object and makes no attempt to teach about objects. The attitude is simply, "lets have some fun writing Java programs." The real instruction about objects and classes comes in Chapter 3 where Slack uses a *Write and Use* approach with an Employee class. Much less interesting than turtles that can draw.

4.2.4 Objects Early: Write, then Use (4)

Jaime Niño and Frederick Hosch have staked out lonely territory in *An Introduction to Programming and Object-Oriented Design using Java* (Wiley, 2002). They introduce the concept of objects (without programming examples) and then proceed to spend chapters 2, 3, 4 and 5 in analyzing, designing and implementing their first class. Students finally write a program to use the class at the very end of chapter 5. But even here, it is merely a small test program to verify that some aspects of the class work correctly. Testing is unappealing in the best of circumstances and should not be the basis of the very first program.[4]

As a textbook for students, I do not think this book is a good choice. However, as a text for teachers, it has much to offer. It includes some of the clearest and most careful use of language in any text and its treatment of Model-View-Controllers to implement graphical user interfaces is exemplary. They have a clear and consistent emphasis on software engineering principles, and I found their linkage of repetition

---

[3] Horstmann actually has two textbooks in the introductory market. *Computing Concepts* is more clearly aimed at CS1. The early chapters of *Big Java* are similar to *Computing Concepts*. Later chapters include many advanced topics not covered in *Computing Concepts*. Unless otherwise noted, comments apply to *Computing Concepts*.

[4] I have been told by marketing representatives that the associated lab manual encourages students to write programs and use objects earlier, but this is not set forth in the text itself.

with collections to be a very interesting idea. Of all the introductory textbooks on my shelves, I felt that I personally learned more from Niño and Hosch than any of the other authors.

4.2.5 Objects Superficially Early (5)

This category includes three textbooks:

- *Java Gently: Programming Principles Explained*, by Judith Bishop (Addison Wesley, 1997)
- *Introduction to Java and Software Design*, by Nell Dale, Chip Weems, and Mark Headington (Jones and Bartlett, 2001)
- *Problem Solving with Java*, by Elliot Koffman and Ursula Wolz (Addison Wesley, 1999)

This category is not discussed in the section on object-oriented pedagogy, above. It arose from examining textbooks that used objects early but did not seem to impart a significant understanding of objects to the student. These textbooks use objects as a means to an end, such as getting input from the user, rather than attempting to actually teach about objects early in the textbook. These textbooks come across as scattered in their early chapters, never really settling down to focus on a topic for the students to learn. I believe they are pedagogically flawed and inferior to many of the books in groups (1), (2), (3), and (4).

4.2.6 Objects Late (6)

The seven books in the Objects Late category are:

- *Java: An Introduction to Computing*, by Joel Adams, Larry Nyhoff, and Jeffrey Nyhoff (Prentice Hall, 2001)
- *Java for Students*, by Douglas Bell and Mike Parr (Prentice Hall, 1999)
- *A First Book of Java*, by Gary J. Bronson (Brooks/Cole, 2002)
- *Java: How to Program*, by H.M. Deitel and P.J. Deitel (Prentice Hall, 1999)
- *Java: A Framework for Programming and Problem Solving*, by Kenneth A. Lambert and Martin Osborne (Brooks/Cole, 2002)
- *Java: An Introduction to Computer Science and Programming*, by Walter Savitch (Prentice Hall, 1999)
- *Java: an Object-Oriented Language*, by Michael Smith (McGraw Hill, 1999)

These books represent what was typical of the first Java textbooks: putting objects approximately where a Pascal textbook would have placed records. As discussed earlier, students using these textbooks have a significant risk of not grasping the central paradigm in an object-oriented program simply because it comes so late the course and students have so little practice writing programs that use objects.

4.3 Recommendations from Others

Robots has not been developed in a vacuum. During its development I have talked freely with colleagues at UW and elsewhere, presented papers, participated in panels, and given workshops[5]. In all of these settings, the pedagogy and the implementation have been enthusiastically welcomed.

---

[5] Becker, Byron Weber (workshop presenter). Objects for Learning to Program with Java, *The Proceedings of the 34rd SIGCSE Technical Symposium on Computer Science Education*, Feb. 19-22, 2003. Page 408.

Becker, Byron Weber (workshop presenter). Teaching CS1 in Java with Karel the Robot, *The Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education*, Feb. 27-Mar. 3, 2002. Page 419.

Becker, Byron Weber. Teaching CS1 with Karel the Robot in Java, *The Proceedings of the Thirty-second SIGCSE Technical Symposium on Computer Science Education*, Charlotte, North Carolina, Feb. 21-25, 2001. Pages 50-54.

Becker, Byron Weber (panel moderator). Polymorphic Panelists, *The Proceedings of the Thirty-second SIGCSE Technical Symposium on Computer Science Education*, Charlotte, North Carolina, Feb. 21-25, 2001. Pages 410-411.

Michael Panitz is one of the people with whom I have talked. He became so excited about the approach that he asked to use the textbook, even though it isn't finished, beginning with the Fall 2002 term. His letter of recommendation is attached. In it, Mike explains why he finds the pedagogy so attractive for his students and says this "is the only book I've ever seen that has a sensible objects-first approach to programming instruction…. The book proceeds effectively, naturally builds on this strong foundation, leading the students to a solid understanding of object-oriented programming."

Sandy Graham is a colleague who has also provided a letter of recommendation. She has taught more than 500 students using the robots metaphor. More importantly, Sandy has talked with many other educators in her role as Waterloo's liaison to high school computer science teachers. Both locally and as far away as South Africa, Sandy regularly encounters people who want to know what Waterloo is doing – and who immediately become excited by the possibilities of following in our footsteps.

Steve Pfisterer is a local teacher who became so excited about our approach that he audited my class to see how it works in practice. His observations about the fun students and instructors are having and how the approach lends itself to active learning are also attached.

Finally, I presented the approach in workshops at the 2002 and 2003 SIGCSE conferences, with very positive reviews. All five referees for the 2003 conference put it in the top category, with comments such as

"It is good to emphasize, as he does, 'client-view first.'",

"Last year's reviews were very positive; it's good to stick with a winner.", and

"valuable topic; previous reviews were high; many people could benefit from this approach."

5 Development Schedule

Robots is more than 50% complete now. I anticipate making significant progress on the remaining chapters between March and June, 2004. I believe it would be realistic to have selected institutions test *Robots* in Fall 2004 and, depending on the amount of change required, to have it ready for adoption by Fall 2005.

6 Potential Reviewers

The following people are familiar with using robots to teach beginning programmers and would be good candidates to review *Robots*.

Rich Pattis, pattis@ux12.sp.cs.cmu.edu, wrote the original *Karel the Robot* book upon which my work depends for inspiration. He currently teaches at Carnegie Mellon University.

Stuart C. Shapiro, shapiro@cse.Buffalo.EDU, teaches at University at Buffalo, The State University of New York. Stuart somehow became aware of my work and requested use of the software for one of his research projects.

Peter Brusilovsky, peterb@mail.sis.pitt.edu, teaches at the University of Pittsburgh. He has experimented with using robots to teach and recently had a paper accepted by *Interactive Technolgy & Smart Education* which provides evidence that my approach of integrating robots into the course is generally superior to the usual approach.

Eugene Wallingford, wallingf@cs.uni.edu, informally spearheads a group of educators exploring "Elementary Patterns." *Robots* builds on this work by identifying the recurring patterns in the text and discussing them in an identified section within each chapter. He teaches at University of Northern Iowa.

Becker, Byron Weber (invited workshop participant). Karel the Robot as an Object Example, *OOPSLA '99 Workshop Report: Quest for Effective Classroom Examples*, Denver, Colorado, Nov. 1-5, 1999.

7 Vita

Byron Weber Becker has been a faculty member in the School of Computer Science at the University of Waterloo, Waterloo, Ontario, since 1991. Prior to joining UW's faculty, he was a software developer for Open Text, one of the early dot-com startups that is still doing well. He has also developed software for the insurance and manufacturing industries. Since coming to Waterloo, Becker has focused on teaching and advising students.

Becker earned a Bachelor of Arts (Mathematics) from Goshen College, Goshen, Indiana. Goshen is a small, liberal arts college that has been cited by *U.S. News and World Report* a number of times for its quality education and innovative international education program. He earned a Masters of Mathematics (Computer Science) from the University of Waterloo. Waterloo has enjoyed the top spot for the last ten years in *Maclean's* magazine's survey of university reputations. The university is widely known for its pioneering work in cooperative education and currently has more than 60 faculty members in the School of Computer Science.