2001

# JKarelRobot: A Case Study in Supporting Levels of Cognitive Development in the Computer Science Curriculum

Duane Buck
*Otterbein University*, DBuck@otterbein.edu

David J. Stucki
*Otterbein University*, DStucki@otterbein.edu

# JKarelRobot: A Case Study in Supporting Levels of Cognitive Development in the Computer Science Curriculum

**Duane Buck and David J. Stucki**
**Otterbein College**
**Mathematical Sciences Department**
**Westerville, OH 43081**
**{DBuck, DStucki} @otterbein.edu**

## Abstract

We introduce a new software tool, JKarelRobot, for supporting an Inside/Out pedagogy in introductory programming courses. Extending the original conception of "Karel the Robot", with Bloom's Taxonomy of Educational Objectives as a guiding principle, we have provided a mechanism for designing exercises that are cognitively appropriate to the developmental levels of our students. JKarelRobot is platform independent (written in Java) and language/paradigm independent, supporting Pascal, Java, and Lisp style environments.

## Keywords

Inside/out Pedagogy, Software Tools, Karel the Robot, Bloom's Taxonomy, CS1, CS2.

## 1 Introduction

It has become patently clear in recent years that the SIGCSE community is embroiled in a Kuhnian [7] crisis of pedagogical paradigms. As he described in *The Structure of Scientific Revolutions*, Thomas Kuhn's explanation of the history of science as a sort of punctuated equilibrium[1] theory of the evolution of scientific knowledge can just as easily be seen as a metaphor for the recent history of computer science education. Nowhere is this more obvious than in recent debates surrounding Introductory Programming (I/O in CS1, language for AP CS, objects first/last/whenever, etc). A common underlying difficulty in such circumstances is a lack of common vocabulary for discourse; but even more fundamentally, the inadequacy of any common conceptual framework to both capture pedagogical objectives and account for experiential successes and failures.

In a previous paper discussing our pedagogy [2], we argued that Bloom's taxonomy of cognitive development should be used to shape the computer science curriculum and provide a vocabulary for discourse. Each level in the hierarchy is subsumed by the next level, so that higher order functioning requires by necessity the lower level skills. A summary of these levels (adapted from [5]) can be found in the Appendix. It is our belief that Bloom provides exactly the conceptual framework that is needed for our community to escape to a period of Kuhnian normalcy.

As a case study of the application of these principles to Computer Science Education we present a new educational software tool called JKarelRobot, developed by the authors. It greatly expands upon the groundbreaking work of Richard Pattis's "Karel the Robot" environment [8]. By incorporating both new language features and support for novel types of exercises into the Karel environment, we have strengthened the pedagogical efficacy of this tool for CS1. JKarelRobot allows the instructor to present targeted concepts to students with neither the syntactical baggage nor the complexities of a real programming environment.

## 2 Conceptual Framework

Our Inside/Out pedagogy involves an incremental, graduated exposure to complexity and structure based on the levels of cognitive development described by Bloom [1].

Inside/out puts students in the context of an overall application design in which they, while working on mastering one level, can glimpse the more advanced concepts present in the layered interface. They are asked to design and code algorithms, first using only language primitives and later some simple library calls. In the context of the design, they are given complete specifications to which their algorithms are expected to conform. Following Bloom, we have now added exercises that are even more "inside" as a starting point.

In our experience, mismatching the lesson to the level of cognitive development of our students results in disaster. Lab assignments, in the form of writing a complete program (Synthesis), given before the students have achieved more

[1] Interestingly, Peter B. Reintjes has suggested the concept of punctuated equilibrium in Darwinian evolution as a metaphor for software engineering practices. [9]

primitive levels, leaves them overwhelmed, uncertain of how to begin, and grasping at the air. Often, this leads them to the self-destructive tendency to do experimental programming, where they just randomly throw things in to see if it helps. It is the students' lack of comprehension of the statements in isolation that keep them from being able to mentally simulate the process, thus encouraging the random walk through code space. Experimental programming has proven to waste a huge amount of students' time, in our experience, with little actual learning. *This is very different from the isolated and controlled experimentation afforded by interpretive environments, where complexity is harnessed and cause/effect is easily identified.* Providing support for building the levels of cognition is the major objective of our study and has helped us shape our software tool.

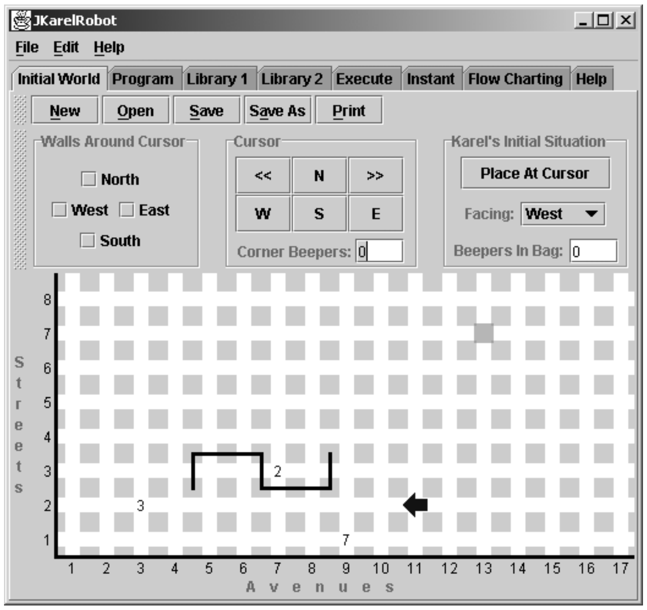
A recent misunderstanding on the SIGCSE listserv (on the part of one of the authors) as to how the word *design* is used by the Teach Scheme [4] project anecdotally illustrates the need for a common language for describing pedagogical methods.

In [2] we took design to refer to *systems design,* the second step of the software development life cycle. Felleisen, on the other hand, uses the word design to refer to a problem solving process that can be used in the small but scales to "software design" in the large. So, in the light of our paper "Design Early Considered Harmful," when Felleisen claimed that "what matters first is design," there was an apparent conflict. After more than a week of exchanging email we discovered that we are fundamentally in agreement on pedagogy for CS1. The semantic subtleties of the word design might have been less of an issue if we had already shared a common conceptual framework like Bloom.

## 3 Karel the Robot

Our software tool extends the environment of Karel the Robot. For those unfamiliar with Karel, here is a brief summary. Karel's world is discrete, and as such he can only exist at discrete coordinates. The vertical coordinates are called streets and they run east to west, while the horizontal coordinates are called avenues and run north to south. He is also only able to face in one of the four cardinal compass directions (north, south, east, west). The streets and avenues are enumerated by the positive integers in a fashion similar to that of quadrant I of the Cartesian plane. There are impenetrable barriers to the south of 1st street and to the west of 1st avenue. In addition to Karel himself there are two types of objects found in his world. Beepers are small devices that emit a faint noise and are also only found at the intersections of streets and avenues. There are also walls that come in one-block segments and are situated halfway between two intersections, centered on the street or avenue they bisect.

Karel is also equipped with a variety of peripheral devices. He has three video cameras, one facing forward, one to the right, and one to the left. He is blind to the rear. The cameras have a limited range, however, and only permit Karel to see just less than one block in any direction. So he is only able to detect the presence of a wall if he right up against it (half a block away). Karel has a compass that tells him which direction he is facing. He has a microphone that is able to hear beeper noises, but it is also limited to hearing beepers at his current intersection. Karel has a mechanical arm and a bag. The arm can pick beepers up off the ground and put them in the bag and take them out of the bag and put them on the ground. Figure 1 shows how the world is visualized in JKarelRobot:


JKarelRobot
File Edit Help
Initial World Program Library 1 Library 2 Execute Instant Flow Charting Help
New Open Save Save As Print
Walls Around Cursor
North
West East
South
Cursor
<< N >>
W S E
Karel's Initial Situation
Place At Cursor
Streets
Avenues

Corner Beepers: 0

Facing: West

Beepers In Bag: 0

**Figure 1: Karel World Editor**

The power of Karel the Robot as a teaching tool lies, we think, in its being able to strip away details that are not important to the concept being taught and learned. This allows students to more readily build a more intuitive conceptual model. The unnecessary details eliminated in this case are variables, replacing such an abstract notion of state with the "state of the world" that Karel occupies. It is this visual state that Karel's program manipulates. This makes the state more real to our students than collections of alphanumeric values stored in aliased memory locations.

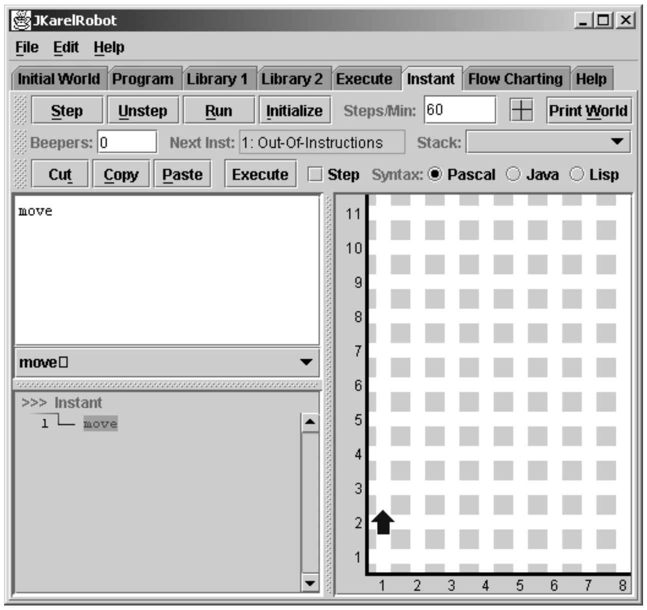
Karel the Robot has proven to be an excellent teaching tool and has been used widely, but it has significant limitations. One limitation, in our experience, is that it does not provide direct support for the Knowledge, Comprehension, and Application levels. This is because students are forced immediately to the Synthesis level by the software. (This is not what is encouraged by Pattis's book, but is effectively what happens.) Also, unfortunately, Karel the Robot is only used for a short time before the students outgrow it. So we have extended the basic Karel language and environment in two ways: (1) to support more directly the primitive levels of cognitive development, and (2) to teach more concepts and support more of the curriculum.

Realizing the heterogeneity of the computer science education community, JKarelRobot was written in Java, and can therefore be deployed on virtually any platform. But, perhaps more importantly, it supports teaching with Pascal, Java, or Lisp syntax, so that it can be used virtually regardless of the platform or teaching language used by a particular curriculum.

## 4 Building levels of cognition

During several years of using Karel the Robot in computer literacy courses, we encountered more than a few students that struggled with mastering the concepts. At the lowest level, some students did not fully grasp the meaning of the graphical display that attempted to visualize Karel in his world. The display was too abstract. One method of overcoming this cognitive gap was to smoothly animate Karel's moves and turns, so that the students see the actions as they are carried out, not just the beginning and ending states of each instruction. This bridges the gap between the display and the students' personal experiences of moving about in the world, and brings Karel to life for them.

Another problem for the students was the necessity of writing a whole program as the first experience with the language used to control Karel. This was too large a conceptual step. Although our tutorial started by having them do very simple tasks, some students had significant difficulties conceiving of how to develop a whole program without fully understanding the primitive elements in isolation.



```
move
```

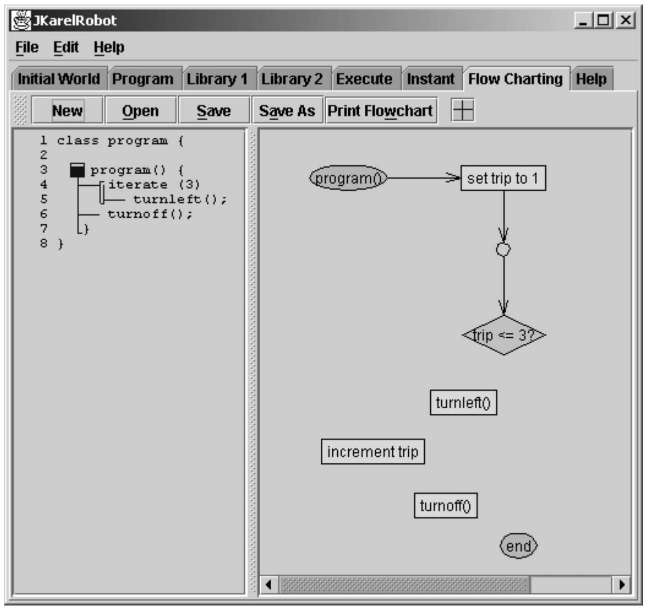
```
>>> Instant
  1 └─ move
```

Figure 2: Karel's Instant Window

We are not alone in these kinds of experiences. The members of the Teach Scheme [4] project at Rice also propose teaching first not by writing a stand-alone program, but by issuing single expressions to an interpreter. They propose that the beginning language be Scheme, because the language is more "algebraic" than imperative languages. But after extended discussion they also state that the nature of the Scheme interpretive environment makes it easy and fun for the students to learn the various statements through experience, that is, typing them in and seeing what happens. We think that the interactive environment is a major factor in their success, not necessarily the language that they use.

The utility of an interpretive environment for learning the concepts of programming and algorithm development is an important insight we gained from the Teach Scheme project. We enhanced our JKarelRobot software, adding an interpretive mode (see Figure 2). In the "Instant" window the students can type in one or more statements and see Karel carry them out immediately. If a statement is not syntactically correct, the students get immediate feedback.

When we examine Bloom's taxonomy for guidance, we find that the interactive environment provides graduated experiences through the cognitive levels. At the Knowledge level we have the statements and their syntax. At the Comprehension level is the capability of predicting what the statement will cause Karel to do in his current situation. We provide direct support for an interactive exercise at the Comprehension level. As they single step through the program the students are asked to predict the next statement that will be executed. They are given a score at the end of the run. At the Application level the students can apply a statement to achieve a desired effect. Once we get to the Analysis level, the compiled mode may also be used effectively, although breaking control structures down may be a good interactive exercise at the Analysis level.



```
class program {

  program() {
    iterate (3)
      turnleft();
    turnoff();
  }
}
```

Figure 3: Flow Chart Exercise

We feel that it is important for our students to learn the language of classical flow charts, in order to be fully conversant in the field. However, because flowcharts have largely been discredited pedagogically as a design tool, we instead utilize them in an exercise at the Analysis level. The task is to translate a given program from language control structures into flowcharts (i.e., the reverse direction from the historical usage). Doing the translations reinforces the meaning of the control structures for them, and particularly helps with nested control structures. Direct support for this type of flowchart exercise is provided within JKarelRobot (see Figure 3).

We have been teaching Java using the jGRASP editor/development environment from Auburn University [3]. jGRASP annotates the code with Control Structure Diagrams (CSDs), which give visual cues to the dynamic behavior of the code. It is multilingual in that the same basic symbols can be used to annotate any imperative language. It also supports ADA95, C, and C++. We have found that its availability is important for our students to quickly *comprehend* the meaning of programs they develop (especially nested control structures) or code they are asked to read. However, the students are given only a brief introduction to the symbols, and they mainly learn to read the annotation on their own, in the process of doing other exercises. We support CSDs in JKarelRobot so that the students become accustomed to utilizing these powerfully evocative annotations early on.

In another Analysis-level exercise, students are asked to examine a given program, locate a minor error that keeps it from meeting its specification, explain the error, and fix it. A related assignment is to give the students a program, along with its specification, and then ask them to implement a program with slightly different specifications. The Synthesis level is where we formerly incorrectly placed students before they developed through the more primitive cognitive levels. However, once they have worked through earlier stages of the curriculum and have the cognitive background, they can be asked to write a program that meets a specification. At the highest level, Evaluation, we have students compare two alternative implementations that meet a common specification. One may ask "what's the difference in what we're proposing at the higher cognitive levels when one compares it to teaching with a traditional programming language?" The beauty of Karel is that the state of the world that Karel manipulates is more familiar to beginning students than the more abstract concept of a variable, so we have less interference with learning the control structures.

## 5 Building beyond Karel the Robot in CS1

Students don't just make one pass through the cognitive levels, winding up enlightened! For each new topic, they must start over and progress through the levels [6]. We can make an analogy here with our experiences with the systems development process. The waterfall model has been largely discredited, while the current trend is toward iterative, incremental development with a lot of intermediate builds, each adding more capabilities. The latter has proven to be much more effective.

The first thing we added to Karel the Robot was Boolean expressions. Students have found it frustrating not to have a way to express the natural concepts of 'and' and 'or,' when many problems naturally need to utilize compound conditions in their solution. The students are therefore motivated to learn Boolean expressions, and already have some of the knowledge and comprehension needed to use them, from their background using natural language. They can apply Boolean expressions to the problem at hand, and then move on to the other cognitive levels.

The question arises as to how much we should decompose the topics and isolate each topic through the levels of cognitive development. One runs the risk of alienating the students by making things too artificial, too removed from the world. This is a classic debate among educators. Our practice is to not necessarily go through all the cognitive levels of a topic at one time, but to intermix the topics and return to topics that have been suspended, similarly to Howard et al. [6].

We looked for ways we could support more concepts from the curriculum, which would incrementally build upon the lessons learned with Karel. As an example, being given a parameter, that is, an unknown that is supplied at a later time, is fundamentally simpler than having a variable that can dynamically take on different values during the operation of the algorithm. So, instead of adding variables to Karel, with the extra baggage that entailed, we added the ability to specify a parameter to a procedure. This lets us cover procedure parameters (a symbol being bound to a value) without the confusion of variables that are usually present when the concept is taught. The student sees the actual parameter being bound to the formal parameter as the execution progresses.

We decided that integer parameters would be the most useful. Karel the Robot already had positive integer constants (used in definite iteration). To deal more fully with integers, we added the integer functions succ(i) and pred(i). The student can from these create integer expressions. An integer expression can be used as an argument to the built-in Boolean function iszero(i) (which we also added), or as the count for definite iteration. Here, we do not offer infix notation, as we do for Boolean expressions because we feel that is better delayed until variables are introduced in the target programming language.

## 6 Future Work

Another application of JKarelRobot we plan to explore this year is in teaching recursion. All three language modes allow recursion, and we provide support for visualizing recursion. Recursion provides natural exercise involving parameters (although recursion could be naturally introduced at this point without using parameters as well). Even though recursion was originally added to JKarelRobot to support teaching in the Lisp syntax, it has led us to explore an early introduction to recursion in CS1 using Java syntax.

Another way we enhanced the software was to add additional languages to control Karel. We had two motivations here. One was to provide a closer match to the target language of the first course, so that JKarelRobot could support more curricula. However, providing more languages can also provide support for a Languages course. We are currently investigating using JKarelRobot in this way at Otterbein. JKarelRobot supports Pascal, Java, and Lisp syntax. This provides an opportunity for students to learn additional notations without having to learn a new development environment at the same time, thus driving home the commonalities of structure. In fact, a good multi-lingual exercise is translation of an algorithm from one language to another.

Modules can be developed in all three languages and calls can be made between them. The instant window supports all three languages, and may call upon compiled modules of any of the languages. For Pascal and Java syntax, parameterized procedure calls are statically checked in the compiled mode. For all three languages, the calls are dynamically checked in the interpretive mode. This allows students to experience the difference of dynamic verses static checking.

## 7 Conclusion

We have recognized that the natural tendency to teach according to the structure of one's own understanding runs contrary to established models of cognitive development. Bloom's Taxonomy has provided a basis for establishing a more efficacious pedagogy, particularly helping to identify topics, exercises, and assignments for CS1 and CS2. Emphasizing a hierarchical progression of skill sets and gradual learning through example, our approach advocates teaching software development from the inside/out rather than beginning with whole programs. With JKarelRobot, we provide further support for our approach. We have also developed web-based materials supporting our approach in Java, and have made them freely available to the community at http://math.otterbein.edu/JKarelRobot.

## 8 Appendix: Bloom's Taxonomy

### 8.1 Knowledge

The remembering of previously learned material. This may involve the recall of a wide range of material, from specific facts to complete theories, but all that is required is the bringing to mind of the appropriate information.

### 8.2 Comprehension

The ability to grasp the meaning of material. This may be shown by translating material from one form to another (words to numbers), by interpreting material (explaining or summarizing), or by estimating future trends (predicting consequences or effects).

### 8.3 Application

The ability to use learned material in new and concrete situations. This may include the application of such things as rules, methods, concepts, principles, laws, and theories.

### 8.4 Analysis

The ability to break down material into its component parts so that its organizational structure may be understood. This may include the identification of parts, analysis of the relationship between parts, and recognition of the organizational principles involved.

### 8.5 Synthesis

The ability to put parts together to form a new whole. This may involve the production of a unique communication (theme or speech), a plan of operations (research proposal), or a set of abstract relations (scheme for classifying information). Stresses creative behaviors, with major emphasis on the formulation of new patterns or structure.

### 8.6 Evaluation

The ability to judge the value of material (statement, novel, poem, research report) for a given purpose. The judgments are to be based on definite criteria. These may be internal criteria (organization) or external criteria (relevance to the purpose) and the student may determine the criteria or be given them.

## References

[1] Bloom, B.S., et al. *Taxonomy of Educational Objectives: Handbook I: Cognitive Domain*, Longmans, Green and Company, 1956.

[2] Buck, D.B and Stucki, D.S. Design Early Considered Harmful: Graduated Exposure to Complexity and Structure Based on Levels of Cognitive Development. Proceedings ACM SIGCSE Symposium, 2000, 75-79.

[3] Cross, J.H., Maghsooloo, S., and Hendrix, T.D. The Control Structure Diagram: An Initial Evaluation. *Empirical Software Engineering*, Vol. 3, No. 2, 131-156, 1998.

[4] Felleisen, M., et al. *How to Design Programs: An Introduction to Programming and Computing*, MIT Press, 2001.

[5] Gronlund, N.E. and Linn, R.L. *Measurement and Evaluation in Teaching*, 6th ed., Macmillan, 1990.

[6] Howard, R.A., Carver, C.A, and Lane, W.D., Felder's Learning Styles, Bloom's Taxonomy, and the Kolb Learning Cycle: Tying It All Together in the CS2 Course, Proceedings ACM SIGCSE Symposium, 1996, 227-231.

[7] Kuhn, T.S. *The Structure of Scientific Revolutions*, 3rd ed., University of Chicago Press, 1996.

[8] Pattis, R.E. *Karel the Robot: A Gentle Introduction to the Art of Programming*, 2nd ed., Wiley, 1995.

[9] Reintjes, P.B. Prolog for Software Engineering, 1994 International Conference on the Practical Applications of Prolog, http://www.logic-programming.org/people/Reintjes_Peter/se94.htm