S$\boxed{\text{oak}}$ed in Java: A Book Proposal

Dan-Adrian German

August 28, 2003

Teller “Simplicity is of interest only as an opposite to unnecessary complication; taken by itself, it is nothing more than monotony. [. . .] In none of this discussion do I mean to imply that the world or life or the future can be simple. A more modest and realistic claim is made: to pursue simplicity in life, in the world, for the future, is a most valuable enterprise.” ([1], pp. 14, 19)

Hamming “We are rapidly approaching an infinite amount of knowledge in the form of results, both in what is known and in what is useful [. . .] We must abandon the methods of retrieval for those of regeneration.” ([2], pp. xv-xvi)

Steele “This is the nub of what I want to say. A language design can no longer be a thing. It must be a pattern— a pattern for growth —a pattern for growing the pattern defining the patterns that programmers can use for their real work and their main goal. [. . .] The Java programming language has done as well as it has up to now because it started small. [. . .] It has grown quite a bit since then. If the design of the Java programming language as it is now had been put forth three years ago, it would have failed—of that I am sure. Programmers would have cried, “Too big! Too much hair! I can’t deal with all that!” But in real life it has worked out fine because the users have grown with the language and learned it piece by piece, and they buy in to it because they have had some say in how to change the language.” ([3], pp. 233-234)

Dijkstra ”Computing’s core challenge is how not to make a mess of it. [. . .] [B]ecause we are dealing with artifacts, all unmastered complexity is of our own making; there is no one else to blame, and so we had better learn how not to introduce such complexity in the first place[1]. [. . .] Also, we know that we can only use a system by virtue of our knowledge of its properties and similarly must give the greatest possible care to the choice of concepts in terms of which we build up our theories: we know we have to keep it crisp , disentangled, and simple if we refuse to be crushed by the complexities of our own making. But complexity sells better, and so the market pulls in the opposite direction. I still remember finding a book on how to use “Wordperfect 5.0” of more than 850 pages, in fact a dozen pages more than my 1951 edition of Georg Joos’s *Theoretical Physics!* It is time to unmask the computing community as a Secret Society for the Creation and Preservation of Artificial Complexity.” ([4], pp. 61-64)

[1][A]ll through history, simplifications have had a much greater long-range scientific impact than individual feats of ingenuity.

1 The Quest for Simplicity

A few years ago when Java was introduced one of the major debates was whether objects should be taught early in the course, or a bit later. In the time spent since, a growing consensus has slowly but irreversibly endorsed the object-first approach. Not much remains to be said about what was once a relatively hot pedagogical debate. A second, and just as serious, difficulty remains though and is surfacing in public awareness only now. What (and how much of it) should an introductory programming course using the Java programming language cover?

This is not a new question. There are now hundreds of introductory programming books using Java and they can all be counted as legitimate answers. A growing consensus of *inertia* (whose roots can be easily traced to the original excitement generated by applets when Java was first introduced in 1994) seems to endorse the idea that textbooks should try to keep up with the plethora of new features. Like the character Fiona in DreamWorks' 2001 animated feature `Shrek` we say: "This is *not* the way it is supposed to happen.[2]" But this is a process that cannot be stopped easily; and the key is to offer alternatives.

2 The Seeds of Growth

> "If you give a person a fish, he can eat for a day. If you teach a person to fish, he can eat his whole life long. If you give a person tools, he can make a fishing pole—and lots of other tools! He can build a machine to crank out fishing poles. In this way he can help other persons to catch fish" ([3], p. 233)

> "If we want to start new things rather than trying to elaborate and improve old ones, then we cannot escape reflecting on our most *basic* conceptions." ([5], p. 17)

The Java programming language is a general-purpose, concurrent, class-based, object-oriented [3] language. Its elements are: (abstract) classes, methods, inheritance, and interfaces. Only when this core set of features is mastered can the student graduate to the study of its first few essential system-provided classes: threads, exceptions, graphics and applets. Few books ([7], [9]) have stopped here though. Most drench their readers into a sea of details, when the goal should be to teach navigation, and the apt use of navigational tools.

[2]In one of the movie's more characteristic scenes, when she attempts singing to a bird, clearly mocking the classic cutesy duet in *Snow White* (or was it *Cinderella*?) the bird strains so hard to keep up with her high notes that it explodes in a hail of feathers.

[3]This means that the language itself is small and has a clear semantic model, but promotes a discipline of programming that facilitates customized, and individual growth.

3 The Core Language

This is an introductory book for absolute beginners. It is the first in a series of two volumes. This volume concentrates on the basic elements of the language. In order, it covers the following topics: algorithms, values, types, expressions, variables, methods, user-defined classes, conditional statements, recursion, iteration, abstract classes and interfaces, applets, graphics and event-handling. Basic data structures are also covered[4]. To give the student a sense of accomplishment (and to prepare her for the study of the second volume[5]) the book gradually develops two projects of moderate size. Variants of the Alien Landing Game ([14]) and a recent version of Pengo (IceBlox, [15]) are developed.

4 The Book's Approach

> "What do we mean by scaffolding in the context of student research in school? There is no appropriate (educational) definition in a dictionary. The term is relatively new for educators, even though the concept has been around for a long time under other names. We tend to think of *structures* thrown up alongside of buildings to support workers in their skyward efforts.
>
> Structure *is* the key word. Without clear structure and precisely stated expectations, many students are vulnerable to a kind of educational *wanderlust* that pulls them far afield. The dilemma? How do we provide sufficient structure to keep students productive without confining them to straight jackets that destroy initiative, motivation and resourcefulness? It is, ultimately, a balancing act.
>
> The workers cleaning the face of the Washington Monument do not confuse the scaffolding with the monument itself. The scaffolding is secondary. The building is primary. The same is true with student research. Even though we may offer clarity and structure, the students must still conduct the research and fashion new insights. The most important work is done by the student. We simply provide the outer structure." ([13], Ch. 19)

Scaffolding is the process by which someone organizes an event that is unfamiliar or beyond a learner's ability in order to assist the learner in carrying out

[4] One- and two-dimensional arrays, arrays of objects, vectors, and hashtables.

[5] The second volume is focused on the teaching of patterns in the context of developing a larger abstraction (a game engine) for the development of multiplayer games in Java. It covers RMI, sockets, objects serialization and XML-RPC in the context of Java and Macromedia Flash MX. The main focus of the second volume is on building the larger abstraction (the game engine) and to expose the student to a collection of patterns ([8]) to grow the language into ever more powerful abstractions. The reader should not be misled by the emphasis on networking and distributed computation. Conceptually, as the book shows, a simple transformation can be used to turn a standalone simulation that involves multiple autonomous agents, competing or cooperating into a shared world, into a multiplayer networked game.

that event. Learners are encouraged to carry out parts of tasks that are within their ability, and the instructor “fills in” or “scaffolds” the rest. This involves recruiting the learners interest, reducing their choices, maintaining their goal orientation, highlighting critical aspects of the task, controlling their frustration, and demonstrating activity paths to them ([11], [12], [13][6]).

Science educators and teachers know first-hand what inquiry instruction looks like, as practiced in a typical classroom. Current models describe inquiry as a matter of steps or phases conducted in succession or in cycles expressed in terms of expected student cognition. Descriptions of teaching practices to elicit and maintain cognitive engagement have remained at a level of generality that leaves the operational meaning up to the classroom teacher. Teaching practices are typically stated in terms of “engaging students in discussion” or “doing an activity” that causes “cognitive conflict”. To work out the operational form of instruction, a teacher must be skilled in a variety of strategies in order to design instruction that maintains the desired cognitive demands of inquiry while adjusting to the constraints of a typical classroom.

Sometimes the teacher must settle for an approximation of inquiry instruction. As a result, instruction may look less student centered than the accepted view of classroom inquiry implies, and more teacher centered. That is, where students are not functioning sufficiently well with the content or materials for any of a variety of reasons, the teacher must carry more of the burden for organizing the content, raising points for consideration, and planning subsequent steps in the instruction.

The book has been written with these instructors in mind. With them, and their students, to offer them a common base for success.

5 To See The World in A Drop of Ink

Here, now, is the entire book in one short section.

The *whole* book?	Yes. Just give me the highlights.
OK. Why is programming fun?	I don’t know. You tell me.
I think the reason I like it so much is that it gives me a world I can control.	Could be. You a tad insecure?

[6]There are at least eight characteristics of scaffolding: (a) it provides clear directions; (b) it clarifies purpose; (c) it keeps students on task; (d) it offers assessment to clarify expectations; (e) it points students to worthy sources; (f) it reduces uncertainty, surprise and disappointment; (g) it delivers efficiency (“Scaffolded lessons still require hard work, but the work is so well centered on the inquiry that it seems like a potter and wheel. Little waste or wobbling. Scaffolding “distills” the work effort.); (h) it creates momentum (“In contrast to traditional research experiences, throughout which much of the energy was dispersed and dissipated during the wandering phases, the channelling achieved through scaffolding concentrates and directs energy in ways that actually build momentum.”)

No. (Snicker). The laws of nature in that world are published, and knowing them, you can	. . . make things happen to your liking. I agree.
There are *few* limits to what you can accomplish if you just think hard enough.	Not quite like the real world.
The real world?	"... the location of non-programmers and activities not related to programming" ([16], [6] p. xvii)
Very good. What is programming?	Programming is a solution to a problem like this:

> "You are given two different length strings that have the characteristic that they both take exactly one hour to burn. However, neither string burns at a constant rate. Some sections of the strings burn very fast; other sections burn very slowly. All you have to work with is a box of matches and the two strings. Describe an algorithm that uses the strings and the matches to calculate when exactly 45 minutes have elapsed."

That's *parallel* programming[7].	I agree. What $\boxed{\text{is}}$ programming?
Programming is a solution to a problem like this:	That's *logic* programming.

> "A farmer lent the mechanic next door a 40-pound weight. Unfortunately, the mechanic dropped the weight and it broke into four pieces. The good news is that, according to the mechanic, it is still possible to use the four pieces to weight any quantity between one and 40 pounds on a balance scale. How much did each of the four pieces weigh? (Note: You can weigh a 4-pound object on a balance by putting a 5-pound weight on one side and a 1-pound weight on the other)."

Quite true.	Is *programming* related to solving a problem like this?
Where's the problem?	Turn the page.

[7]Burning a string from both ends would make the string last only $\frac{1}{2}$hour, wouldn't it?

Ah! *Sequential* programming.

> "A captive queen weighing 195 pounds, her son weighing 90 pounds, and her daughter weighing 165 pounds, were trapped in a very high tower. Outside their window was a pulley and rope with a basket fastened on each end. They managed to escape by using the baskets and a 75-pound weight they found in the tower. How did they do it? The problem is anytime the difference in weight between the two baskets is more than 15 pounds, someone might get killed. Describe an algorithm that gets them down safely."

A bit conservative, I agree[8].

Do you have more examples?	Sure. But how about programming in Java?
Very well. What is this: 5	A number. Java calls that an `int`.
What is this: 3 + 5	Java calls that an *expression.*
What's the value of this expression? $2 - 3 + 5$	It's different from $2 - (3 + 5)$
Very good. What's the value of $2/3 * 6$	Shouldn't this work the same? $6 * 2/3$
No, and that's the whole point.	Very good. How do you calculate $3 + 5 - 2$
First an 8 gets created.	Where do we store it?

[8]This problem can also be used to illustrate the notion of a *named procedure*—just like the solution to the previous puzzle (using a *balanced ternary system*) can help bring up the topic of arithmetic with positional number systems. Solving the puzzles reveals even more.

I don't know, it hangs around

So things can be built in stages:

```
int result = 3 + 5;
result = result - 2; // gives us 6
```

What's `result`?

A *name*. The name of a *variable*.

What is a variable?

A location with a name (and a type).

What's this?

That's a cupholder.[9]

```
class Pair {
  int x;
  int y;
}
```

Could be a `Point2D`.

Or a `Fraction`.[10]

They *look* similar.

They just *behave* differently.

How do you create a new cupholder?

You say: `new Pair()`

How do you place the cups in?

Easy. Start by giving it a name:

```
Pair a = new Pair();
a.x = 3;
a.y = 5;
```

Then use the name of the cupholder to place the individual cups.

I see...

You could have more than one cupholder.

Indeed.

And you'd be accessing them in the same way.

[9]Did somebody say McDonald's? (We didn't think so.)
[10]Really?

Like this:

```
Pair a, b;
a = new Pair();
b = new Pair();
a.x = 3;
a.y = 5;
b.x = 1;
b.y = -2;
```

How many *kinds* of variables do we have in Java?

Four: local, instance, `static` variables, and also parameters.

Cups are *instance* variables.

Do you understand this?

$$f(x) = x + 1$$

x is a parameter.

Do you understand this?

$$f(x) = \begin{cases} \frac{x}{2} & \text{if } x \text{ is even} \\ 3x + 1 & \text{otherwise} \end{cases}$$

Ah! `if` statements.

Do you understand this?

$$f(x) = \begin{cases} 1 & \text{if } x = 1 \\ x + f(x-1) & \text{otherwise} \end{cases}$$

This one is a loop ... $f(10)$ is 55.

How do you do this in Java?

Ask Alan Kay[11].

Seriously...

Easy. Turn the page.

---

[11]"The ability to start with an idea and see it through to a correct and efficient program is one prerequisite for a great software designer. A second is to see the value of other people's good programming ideas. In 1961 Kay worked on the problem of transporting data files and procedures from one Air Force air training installation to another and discovered that some unknown programmer had figured out a clever method of doing the job. The idea was to send the data bundled along with its procedures, so that a program at the new installation could use the procedures directly, even without knowing the format of the data files. The idea that a program could use procedures without knowing how the data was represented struck Kay as a good one. It formed the basis for his later ideas about objects." ([10], p. 41)

Here's sum as defined above.	Programming uses longer names.

```
int sum(int x) {
  if (x == 1) return 1;
  else return x + sum(x - 1);
}
```

Not bad...	Show me more.
Here's a Fraction	It knows how to add.

```
class Fraction {
  private int num, den;
  Fraction(int num, int den) {
    this.num = num;
    this.den = den;
  }
  Fraction add(Fraction other) {
    return new Fraction(this.num * other.den +
                        this.den * other.num,
                        this.den * other.den);
  }
}
```

You said there were four kinds of variables in Java.	Yes. Local variables and parameters are essentially the same.
You don't need to initialize parameters. The caller provides them.	Local variables need to be initialized.
Instance and static variables should be accessed by first locating the object (or class) that contains them.	Is *that* what this is for?
In this case: yes.	Objects are containers, and classes are containers. Objects can be created on the fly though.
Classes contain the blueprint.	Blueprints can be described in stages.
What is a Horse?	Difficult to describe.

Why?

Too complex. Way too many parts.

Can we simplify?

Sure. Let's abstract away.

```
class Horse {
  void greet() {
    this.neigh();
  }
  void neigh() {
    System.out.println("Howdy!");
  }
}
```

Looks good.

I know.

What's a `Unicorn`?

Oh, that's easy:

```
class Unicorn extends Horse {
  Horn horn;
}
```

Is that all?

Yes. Set union of features.

What's my next question?

Can we have complications?

```
class Unicorn extends Horse {
  Horn horn;
  void neigh() {
    System.out.println("Bonjour.");
  }
}
```

Oh, ya—but that's the beauty of it.

That's how applets[12] work.

Is every `Unicorn` a `Horse`?

Yes[13].

[12] Where `neigh` is `paint` and `greet` is `update` (or `repaint`).
[13] Arrays of objects rely on this premeditated loss of detail.

Is every `Horse` a `Unicorn`?

No.

So what's the outcome of this:

```
Horse a = new Unicorn();
a.greet()
```

`Bonjour.`

Very good.

It used to be `Howdy!` though.

Is the class extension mechanism of use only in *large* programs?

No, take a look at this:

```
class One {
  int add(int n, int m) {
    if (m == 0) return n;
    else return add(n+1, m-1);
  }
}
```

What about it?

Relax. Do you understand it?

I sure do.

Do you, really?

Sure. Here's to prove it:

```
class Two extends One {
    int mul(int n, int m) {
      if (m == 1) return n;
      else return add(n, mul(n, m-1));
    }
}

class Three extends Two {
    int pow(int n, int m) {
      if (m == 0) return 1;
      else return mul(n, pow(n, m-1));
    }
}
```

Super.

Technically that's a reserved word.

Well: nifty, then.

What's an interface?	An element of pure design.

```
interface Multiplier {
  int mul(int n, int m);
}
```

I see...	Wait a minute, there's more to it.

```
class Alpha implements Multiplier {
  public int mul(int n, int m) {
    return n * m;
  }
}

class Beta implements Multiplier {
  public int mul(int n, int m) {
    int result = 0;
    for (int i = 0; i < m; i++)
      result += n;
    return result;
  }
}

class Gamma implements Multiplier {
  public int mul(int n, int m) {
    if (m == 1) return n;
    else return n + mul(n, m-1);
  }
}
```

Do you have other implementations?	Of course, but that's irrelevant.

What's relevant at this point?	What you can do with interfaces.

Patterns[14].	Yes, patterns in action[15].

[14] Patterns are also elements of pure design.

[15] This would be a perfect time to turn the page.

Take a look at this problem[16].	I want to see it.

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class Game extends Applet {
    int i = 0;
    public void paint(Graphics g) {
        this.i = this.i + 1;
        System.out.println("Paint called: " + i);
    }
    public void init() {
        Umpire ump = new Umpire();
        this.addMouseMotionListener(ump);
    }
}

class Umpire implements MouseMotionListener{
            // wearing the uniform...
    public void mouseDragged(MouseEvent e) {
        System.out.println("Ha! You're dragging the mouse.");
    }
    public void mouseMoved(MouseEvent e) {
        System.out.print  ( "Mouse seen being moved at: (");
        System.out.println( e.getX() + ", " + e.getY() + ")");
    }
}
```

How is an interface like a uniform?	A uniform is also void of content[17].
It acts as a signal.	It signals a social convention.
An umpire *must* wear a uniform.	But you need a *real*[18] umpire.

[16]This is the celebrated `Observer`/`Observable` pattern, of course.
[17]This would be a great time to discuss `abstract` classes
[18]Deceiving looks won't do. One must implement the advertised contract.

And an umpire without the uniform won't be recognized by the crowds.	An umpire in his uniform, at home, asleep in his armchair won't be able to even participate in the game[19].
Good. The rest is API[20].	Yes, the rest is Mozart[21].

References

[1] Edward Teller, *The Pursuit of Simplicity,* 1980 Pepperdine University Press

[2] Richard Hamming, *Methods of Mathematics Applied to Calculus, Probability, and Statistics,* 1985 Prentice Hall

[3] Guy L. Steele Jr., *Growing a Language,* Journal of Higher-Order and Symbolic Computation (Kluwer) 12, 3 (October 1999), 221-236

[4] Edsger W. Dijkstra, *The Tide, Not The Waves,* in Beyond Calculation (The Next Fifty Years in Computing) edited by Peter Denning and Robert M. Metcalfe, 1997 Springer-Verlag

[5] Hans Primas, *Chemistry, Quantum Mechanics and Reductionism,* Springer-Verlag (Berlin) 1983

[6] Julie Sussman, *Foreword* to *Programming in MacScheme* authored by Eisenberg, Hartheimer, Clinger and Abelson, 1990 The Scientific Press

[7] Friedman and Felleisen, *A Little Java, A Few Patterns* 1997 The MIT Press

[8] Gamma, Helm, Johnson, and Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Patterns*, 1995, Addison-Wesley Publishing Co.

[9] Ken Arnold, James Gosling, David Holmes *The Java Programming Language* Addison-Wesley, 2000

[10] Dennis Shasha, Cathy Lazere *Out of their minds.* Copernicus, Springer-Verlag, 1995

[11] Wood, D., Bruner, J. S., Ross, G. (1976). *The role of tutoring in problem solving.* Journal of Child Psychology and Psychiatry, 17, 98-100.

[12] Wood, D., Middleton, D. (1975). *A study of assisted problem solving.* British Journal of Psychology, 66, 181-191.

[13] Jamie McKenzie *Beyond Technology: Questioning, Research and the Information Literate School Community* FNO Press, January 2000.

[19] So add it as a listener, to make the calls. (Assign the umpire to the game.)

[20] `http://java.sun.com/products/jdk/1.4/docs/api/overview-tree.html`

[21] And that's from Victor Borge.

[14] Joel Fan, Eric Ries and Calin Tenitchi, *The Black Art of Java Game Programming,* Waite Group Press, 1996.

[15] Kay Hornell, in *Cutting-Edge Java Game Programming,* Sams, 1996.

[16] Steele et al., *The Hacker's Dictionary,* Harper and Row, 1983.