How We Should Teach Programming – How We Should Not Teach Programming

Chris Stephenson, Istanbul Bilgi University

Summary

The results we achieve when teaching programming are not good. Many Computer Science / Computer Engineering students graduate unable to program well. I suggest adoption of some new approaches, emphasising program and data design from the first day of programming teaching. These are contrasted with examples from conventional textbooks in current use. Experience, both in the author's University and elsewhere suggest that this is a fruitful path to follow.

Keywords: Programming, Computer Science Education, Java, C, Scheme.

What is really important in programs? F P Brooks wrote:

> Much more often strategic breakthrough will come from redoing the representation of the data or tables. That is where the heart of a program lies. Show me your flowchart and conceal your tables and I shall continue to be mystified. Show me your tables and I won't usually need your flowcharts, they'll be obvious.[1]

That was in 1975, more than 30 years ago. Brooks derived his conclusions from his experience as the manager of a massive and difficult programming project he led in the 1960s – IBM OS/360. His conclusions remain entirely true today. Brooks is putting two things at the centre of the software design process: the design of the structure and definition of data and the level of abstraction necessary for effective reuse of code. When we look at the programming techniques that are currently gaining in popularity, they all reflect the truth of what Brooks wrote about the importance of data design and abstraction.

The centrality of Data Design appears in many areas., Object Oriented Programming is, among other things a way of attaching code to data structures and putting the code in a subsidiary position in the structure of a program. Model-View-Controller techniques of system design give the determinant role to the design of the Model – that is the data structure and its accompanying semantic definition. In a J2EE system a critical glue role is played by a large mass of XML data – so large that the main role of many Java IDEs is to create, edit and maintain this XML data structure. Service Oriented Architectures again put the syntax and semantics of data, in this case the XML data, communicated between servers and clients, at centre stage in system design.

Likewise, the idea of the need for abstraction, is at the centre of modern developments in computer systems development techniques. As Brooks observes, anywhere we see repetition, we try to eliminate it. Abstraction is the prerequisite for the generalisations that help us avoid rewriting code. These generalisations surface as inheritance in Object Oriented Languages, as Iterators and Collections, as Design Patterns. The struggle to solve the "software problem" involves a collection of techniques centred on various kinds of abstraction.

Do we teach our students this way? No. Not only do we not teach them this way, but, in general, in programming teaching, we teach an approach to programming that makes it *more difficult* for our students to subsequently grasp these concepts.

A symptom of this is "copy-paste" programming. Many of our graduates remain unable to write even quite simple programs from scratch. Their main programming technique is to Google for a solution, then adapt it to the task in hand. I have observed this after interviewing hundreds of Computer Science / Computer Engineering candidates for assistant positions at Bilgi. After filtering CVs to find the best qualified candidates, it is still the case that 90% of the candidates we interview are unable to write a program of a few lines from a simple specification, given a blank piece of paper and a pencil.

I want to argue that we need a different approach, that starts for design and from data. In this article I will refer to our experience at Bilgi University, using a design based approach and the functional language Scheme. This is not, however, a technical question about choice of programming languages for teaching. It is possible to teach programming badly using the most appropriate of all possible languages, and possible, though more difficult, to teach it well using the least appropriate. It is a question about approach.

We are teaching programming badly

Why are our students failing to learn to program? I want to argue that it is because we teach them badly. Worse then that, we teach them bad programming. The first article of the Hippocratic oath states that the doctor should do his patient no harm. The same rule should apply to the teachers of programming.

Let us take an example. One of the programs we often teach students to write fairly early in their education is to sort a list of integers into ascending order. It says little for our approach that we teach them to do an essentially useless job using a method which he we explain very badly and which is essentially inefficient in computing time. [i]

However, a sort is a task of medium difficulty, and a short investigation of how we teach it is revealing. The best place to look for examples for teaching programming is, naturally, in programming text books.

I have chosen to look at examples from one C programming text book and 6 Java programming text books. The sample texts were chosen by simply looking at all the introductory programming books I have been sent free as inspection copies. I then selected those that even mention sorting. It is a symptom of the dumbing down of programming education that even a simple sort is now regarded by many text books as "too difficult". It certainly should not be the first thing we try to teach students, but it is a task of medium difficulty that is a reasonable student programming task.

Java for Students[2] leaves sorting an array of numbers into order as an exercise, and simply warns the reader that this is "not the easiest program to write". This is not the best way to educate programmers. It is our job to show our students how to make writing correct programs easier, not to tell them that it is, in some mysterious sense, "not easy". The recommended method is selection sort, which presents a number of serious programming traps about which the book does not warn.

Java: How to Program[3] performs some interesting self plagiarism. The bubble sort program in this book is a copy, right down to the test data used, of the bubble sort program in *C: How To Program*[4]. In fact the program is not really a Java program at all, but a C program transliterated into Java. And the test data is hard coded right into the program. This encourages the very wrong idea that one test is enough. In particular none of the critical cases are covered by the test data. What about zero length arrays or arrays of length 1? What about arrays in which some elements have equal values? Since these are not tested, there is a real possibility that the program fails when these conditions arise.

Look at this fragment of the Deitel code:

```
// swap two elements of an array
public void swap( int array3[], int first, int second ){
        int hold;   // temporary holding area for swap
        hold = array3[ first ];
        array3[ first ] = array3[ second ];
        array3[ second ] = hold;
}
```

The bizarre juggling that is required to swap two values in the swap method hide many difficulties and raise four questions that an enquiring student should ask:

Why do you pass an array as a parameter and two indices to perform a swap? Why not just pass the two values to be swapped? Why do you need the third variable named hold? Why is the swap method declared public? Why is the array parameter given the name array3? Is it perhaps because the array parameter of the bubbleSort function is called array2?

The answer to the first question involves a deep discussion of values, references and pointers, necessitated by the mutability of Java arrays. The discussion is further obscured by the fact that the program example is directly copied from a book on the C language, in which the pointers can be used to avoid passing the whole array as a parameter. The second is an unfortunate consequence of imperative programming style. The answer to the third question is simple: the function should not be declared public. The code example is wrong. The final question shows that the book gives the student the completely wrong message that we need globally unique names for method parameters.

The version of bubble sort Deitel and Deitel give is not only hard to understand but also involves an enormous amount of unnecessary work. It performs n^2 comparisons to process an array that is already in order. More important, there is no clue whatsoever about how to arrive at the program that is given and no real assurance that the program will work. The example code invites a copy-paste approach and actually teaches students that one test is enough.

---

iThe real need for sorting is for tuples sorted by a key of some sort. If you really wanted to sort a list of integers in order, and if the integers are of fixed length, then there is a O(n) method. If you want to sort a list of n tuples then there exist well known and relatively simple methods with computing time O(nlogn). Insertion sort (and bubble and selection sort) taught the way we usually teach them are not at all easy to understand and are also so inefficient, with a computing time of O(n2) that they are unacceptable for use on problems of any realistic size.

Finally at the end of this section, the authors offer the following pearl of wisdom, also copied directly from *C:How to Program*, where it appears as "performance tip 6.5".

Performance tip 7.2

Sometimes the simplest algorithms perform poorly. Their virtue is that they are easy to program, test and debug. Sometimes more complex algorithms are required to realise maximum performance.

This is an example of a general rule about many of the introductory programming books on the "market". If the author emphasises a point by setting it off from the text, putting a box round it and printing it in a different colour, watch out! What is written in the box in purple, brown, red or another colour is *probably not true*. Let us call this observation the Law of the Purple Box. I will offer a "proof" of this law later in the article.

Simple algorithms may or may not perform better or worse than complicated ones. It is far easier to examine the performance of simply and elegantly expressed algorithms than the performance of complicated algorithms and you are less likely to fall into performance traps with simple elegant algorithms. Merge sort is, in Big O terms, optimal for time performance, but can be explained in a few lines. A good implementation of merge sort is less complicated, and fewer lines of code than Deitel and Deitel's bubble sort.

Java Software Solutions Foundations of Program Design [5]

As in other books, the explanation skates over vital parts of the program where apparently innocent changes will stop the program working. It also assumes that a single test is enough to test a sort program.

Example:

```
public static void insertionSort (int[] numbers){
   for (int index = 1; index < numbers.length; index++){
      int key = numbers[index];
      int position = index;
      // shift larger values to the right
      while (position > 0 && numbers[position-1] > key) //***
         numbers[position] = numbers[position-1];
         position--;}
      numbers[position] = key;}
}
```

Look at the line marked with asterisks. If we change the order of the two values combined with &&, thus:

```
      while (numbers[position-1] > key && position > 0 )//***
```

the program will fail with an array bounds exception. To understand this it is necessary to know that && is not a symmetrical logical operator, but a sequenced test in which the second value is not even evaluated if the first value is false. This a terrible trap lying in wait for any student who tries to write a program in this style without simply copying it. We teach our students to copy and paste.

Java Programming from the Beginning [6]

This book seems like a deliberate exercise in making the sort difficult to understand. A fragment of code to insert a number into an already sorted array is developed. Then it is abandoned to write a monolithic insertion sort that performs insertion in the opposite direction to the original function.

After 5 confusing pages this book gives us another classic example of the Purple Box Rule.

When you're designing an algorithm, don't worry about the special cases at first. Design the algorithm to work in the typical case. After you have completed the algorithm, go back and check to see if it correctly handles the special cases. If it doesn't, add additional test for these cases.

Just in case you missed the point, the purple box is repeated again in black and white in the main text.

In other words, don't waste time and complicate your algorithms by worrying about all the special cases before you've designed the general algorithm.

I hope none of the students who learn programming from this book going on to work writing the programs that control aircraft or nuclear power stations. There is a vast literature on software engineering and the selection of test cases by choosing values on or close to boundary cases. Special cases are also often the termination cases for a program. In other words, the "special cases" are central to the proper design of any program.

The program fragment offered for the insertion sort contains the same && trap as the Lewis and Loftus book.

Introduction to Java Programming [7]

This example shares many of the vices of the others, and is notable for nothing other than its purple box:

Most students will not be able to derive the algorithm on their first attempt. I suggest that you write the code for the first iteration to find the largest element in the list and swap it with the last element, and then observe what would be different for the second iteration, the third, and so on. Fron the observation, you can write the outer loop and derive the algorithm.

The Object of Java Introduction to Programming Using Software Engineering Principles [8]

This book has ambitions to do better, but only manages to confuse. It uses the SimpleList collection and writes the program using an iterator over the collection. The book specifies pre and post conditions for its methods. Nonetheless, the && trap is still there, unexplained.

```
public void insert (int j) {
        start();
        while (!isOff() && j > itemInt())    {
                forth();
        }
        super.insert(new Integer(j));
}
```

Reverse the order of the two values combined with && in the while statement and the program will fail with an error.

Summary

We can summarise this brief survey of sorting in programming text books in this little table:

	type of sort	*1 test is enough*	*pages*	*loc*[ii]	*argue general correctness*	*fragile misleading or inexplicable code*	*purple box*
Deitel C	Bubble	yes it is	3	22	no	yes	Wrong
Deitel Java	bubble	yes it is	4	37	no	yes	Wrong
Lewis	insert	yes it is	6	27	no	yes	Absent
King	insert	test is unnecessary!	5	17[iii]	no	yes	Wrong
Liang	selection	yes it is	3	32	no	no	Wrong
Riley	insert	yes, only one example	5	24[iv]	no	yes	Absent

If we accept standards of proof of the same rigorousness as the standards of testing in these texts, then four positive confirmations and no counter-examples to the “Law of The Purple Box” would leave us ready to accept that the Law is proven.

The common feature of these books is that teach a number of things that are positively wrong. They teach not to test properly. They confuse students about variable scope and concepts like public and private. The extremely fragile program examples in the books in practice teach students to to attempt even to modify working code for fear that it will break. Worst of all, these text books server to break students' self confidence. In fact, random probes beyond the limited scope of this article which all uncover even more examples of the teaching of bad programming suggest that these books form a pathological canon worthy of deeper investigation.

We can teach programming better

At Bilgi we now use a design centred approach. The textbook we use in out first year course is How to Design Programs[9]. For a systematic exposition of the design recipe approach, the reader should refer to the HtDP book, fortunately available freely on the web at www.htdp.org. I will also assume that he reader can pick up the extremely simple syntax of the “student beginner” flavour of the Scheme language supported by DrScheme as we go along, and jump straight in to an example some way through our first term course, round about the tenth week.

To give a flavour of how we interpret the HtDP approach, let us redesign insert sort using it.

Step one in the process is to define our data. We want to sort a list of numbers into order. We need to define our input and our output and to define it precisely; our input is list of numbers, our output is a sorted list of

ii non blank, non-comment, lines containing executable code. Author's layout

iii This is not a complete program, just two fragments of code, not even a method

iv This is not a complete program – only insert is programmed

numbers.

What is a list of numbers? It can be one of two things. Either it is empty, or it is a number followed by a list of numbers.

From this data definition and others like it we can derive a pattern, which we call a template, can be used for many operations on lists. Since the list definition consists of two parts, our list processing program also consists of two clauses. The first clause deals with an empty list, the second with the non-empty case.

```
(define (some-func l)
        (cond
                ((null? l) ...)
                (else ... (first l) ... (some-func (rest l))))))
```

What is a sorted list of numbers? However we try to define it we always end up with three, not two, cases in our definition. A sorted list of numbers is either an empty list, a list of one number, or a pair of a number and a sorted list whose first element is not less than that number.

```
;; data definition sorted-list-of-number
;; sorted-list-of-number is
;; empty
;; OR
;; (pair number empty)
;; OR
;; (pair number sorted-list-of-number) where number <= (first sorted-list-of-number)
```

Using the data definition we can immediately write a test for a sorted list. This can serve as another example of how we can derive a program from a data definition. It will also be useful for testing our sort program.

```
(define (is-a-sorted-list-of-number? l)
        (cond
                ((null? l) true)
                ((null? (rest l)) true)
                (else (and ( <(first l) (first (rest l))
                                (is-a-sorted-list-of-numbers? (rest l)))))))
```

What does our sort program need to do? It needs to consume a list of numbers. We have a template for that. Once we have constructed our template, we need to think how to fill it in so that we can construct a sorted list of numbers.

The template does not write our program for us. We also need to define what the program will do. It has to meet two conditions. It must produce a sorted-list-of-numbers and that list must contain exactly the numbers in the list consumed. Some examples will help and we can use them for testing later. We exercise one of the basic methods of software engineering by checking marginal cases.

Here are a few, expressed as a test to be used on our finished sort program, when we have it.

```
(equal? (sort empty) empty)
(equal? (sort (pair 1 empty)) (pair 1 empty))
(equal? (sort (pair 1 (pair 2 empty))) (pair 1 (pair 2 empty)))
(equal? (sort (pair 2 (pair 1 empty))) (pair 1 (pair 2 empty)))
(equal? (sort (pair 3 (pair 2 (pair 1 empty)))) (pair 1 (pair 2 (pair 3 empty))))
```

Looking at the template, we can see that to fill in the clauses left as ... on the right we need somehow to construct a sorted list from a number and an empty list, which is trivial, and to construct a sorted list from a number and a sorted list, which is not trivial. We can come back to these problems when we have completed the design.

So we have written the design for our sort program and chosen a template. But one clause of our program involves processing another list. This cannot be fitted into our template. So our design rules tell us we need a helper function. We now know the constraints within which we must design the helper function. It consumes a number and a sorted list and produces a sorted list. The only other constraint is that the output must contain all the numbers in the two inputs and only the numbers in the two inputs. In fact, if our helper function satisfies this constraint, it is clear that the main function will also do so.

Again some test examples will help us. Again we express them as tests to be used on our finished program.

```
(equal? (insert 3 empty) (pair 3 empty))
(equal? (insert 2 (pair 3 empty)) (pair 2 (pair 3 empty)))
(equal? (insert 3 (pair 2 empty)) (pair 2 (pair 3 empty)))
(equal? (insert 2 (pair 1 (pair 3 empty))) (pair 1 (pair 2 (pair 3 empty))))
(equal? (insert 1 (pair 2 (pair 3 empty))) (pair 1 (pair 2 (pair 3 empty))))
(equal? (insert 3 (pair 1 (pair 2 empty))) (pair 1 (pair 2 (pair 3 empty)))))
```

Once again we can use our list processing template. Our function has to create a list. We create new lists by

using cons to add a new element to the front of a list. The null case for this function is trivial – if we are adding to a null list then we simply pair the new element to the empty list. In the other case, we are guided by the data definition of a sorted list. If the number to be added is less than the first of the sorted list, then we can simply pair it to the list and produce a sorted list that meets the other constraint. If it is not, then we can pair the first of the list onto the result of inserting the number in the rest of the list. So the template for our insert function looks like this:

```
;; insert number sorted-list-of-number -> sorted-list-of-number
;; purpose: insert a number into a sorted-list-of-number
;; template
;; (define (insert n slon)
;;    (cond
;;       ((empty? slon) ...)
;;       ((<= n (first slon)) ... )
;;       ((else ... (first slon) ... (insert n (rest slon)))))
```

So now have definitions of our data, specifications of our two functions, definitions of test cases and templates for our functions.

Notice that we have not yet written a single line of the code of our actual program. Everything we have is either definition, test or template. All that remains is to fill in the gaps to produce our sort program. Looking at the examples, this is easy.

Here are the filled in functions.

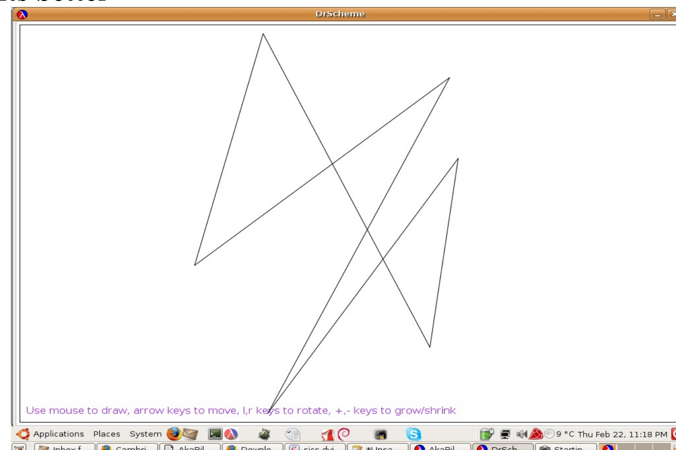
```
(define (insert n slon)
  (cond
    ((empty? slon) (pair n empty))
    ((<= n (first slon)) (pair n slon))
    (else (pair (first slon) (insert n (rest slon))))))

(define (sort lon)
  (cond
    ((empty? lon) empty)
    (else (insert (first lon) (sort (rest lon))))))
```

It may seem that the method outlined is inordinately long for a solution to such a simple problem. However, the aim is to teach students to program, then the way to start is with the same approach to small problems that we will later need to take to larger problems. While this design is not in itself a formal correctness proof for the program, the close tie to the definitions of the value consumed and the value produced makes a program which is, at least, persuasive and can fairly readily be transformed into a formal correctness proof.

In fact, the explanation of insertion sort in this paper is actually shorter than the explanation in several of the books studied. The total length of the program is 9 lines of code. To this should be added 24 lines of testing code. The total is still no more than the comparable text book examples, none of which perform more than a single test. But our program contains much more. It develops validation tests for the input and output of our program. It contains a set of regression tests that will help us to ensure that any modification to the code has not broken it. It contains comprehensive documentation of the inputs and outputs of each function in the program. In other words, it teaches students how to program in a professional way.

In the next section we will look at one example beginner program that uses the same techniques to produce a program which looks far from trivial, though it is simpler conceptually than insert sort.

We can motivate students better



This output is from an example program from the 10th week of the first term of the first year Introduction to Programming course at Istanbul Bilgi University. It is a program that produces a highly motivating graphic result, while the program design is conceptually far simpler than the insert sort example in the preceding section.

The task is to create a program that will draw a polygon on the computer screen where each new node is created by clicking the mouse, where the position of the last node can be adjusted by dragging the mouse and the polygon can be translated, scaled and rotated on the screen using the keyboard.

A data definition directed, abstraction oriented approach immediately reduces what appears to be a difficult programming problem to a relatively simple one that even a novice programmer can tackle. The choice of a suitable Model in a Model-View-Controller implementation is crucial to a successful implementation. The complexity of the problem is massively reduced. Drawing the polygon from the model, the View component, is simple. Then each of the required modifications of the polygon (add a node, move the last node, rotate the polygon, move the polygon and scale the polygon) can all be solved using relatively simple functions independent from the view and independent from each other.

The listing of the program[10], including all of the comments containing the design specifications, show that this task can be accomplished in only 58 lines of executable program code. The code is well laid out and, by virtue of the use of design recipes, extremely well documented, every single one of the functions in the program is instrumented with a battery of regression tests to ensure that modification can be carried out without introducing mystery bugs.. There is no clever trickery. The programming environment has *not* been specially adapted to favour programs of this type. The programming environment provided offers no more than ways of relaying mouse actions and keystrokes to callback functions, together with a function to draw a line between two points. This is a very primitive graphics environment indeed. The advantage comes from the design method, not from the facilities provided by the programming environment.

Experience

The ideas outlined in the first section are not new. The book Structure and Interpretation of Computer Programs(SICP) [11], the text book for the legendary 6.001 course given to all first year engineering students at MIT was an inspiration to this author. How to Design Programs[12] is a book,written to address what the authors see as the deficiencies of SICP as a first year teaching resource. The book tries to demystify and simplify the design process, by providing sets of design recipes that gradually increase in complexity as the book proceeds. We have been using this book in a two-term first year “Introduction to Programming” course at Bilgi University for the last year and a half.

Some scientific studies have been carried out. These show improved performance by students who first take a design oriented programming course using a functional language like Scheme, and then transfer to the popular Java language for their subsequent courses. One interesting additional effect reported is of an evening up of the performance difference between male and female students, with the performance of female students being disproportionately improved by the use of a design based approach. This difference was also reported among high students who received both a conventional Java course and a Program Design oriented coures in Scheme. A majority among all students preferred the design based course, but women students did so by a majority of four to one.[13]

In the US and elsewhere there is external pressure to force programming education to be carried out in a particular language, usually Java. While this does present an educational problem, it is not the main problem. The main problem is the implicit assumption that choosing a language is enough to specify the kind of programming education that we are going to give.

It is, of course, possible to teach programming well using even Fortran. The Elements of Programming Style[14] is an excellent book with all the code examples in Fortran. However, if we are to solve the problem that we graduate many students who are only able to create programs using copy-paste-modify techniques, then we need to shift the emphasis back to design, from the very first day of our programming education.

Conclusion

Teaching to program well is a difficult job. Point-and-click, copy-paste culture makes the development of good programs more difficult. Original, ground breaking programs need to be designed and thought about, not thrown together with a few mouse clicks. The ability to program well is a vital skill in the modern world. If we ignore the problems then programming well will become the reserve of a small elite working in Universities or for a small number of Companies at the technological cutting edge, like Google. If we are to serve our students well, then we need to teach them to think about programs in a deeper way.

1 Brooks, F.P. *The Mythical Man-Month*, Addison Wesley 1995 p 102
2 Bell, P. Parr, M. *Java for Students* Prentice Hall 2002
3 Deitel, H.M. Deitel, P.J. *Java: How to Program* Prentice Hall 2003
4 Deitel, H.M. Deitel, P.J. *C: How to Program* Prentice Hall 1992
5 Lewis, J. Loftus W. *Java Software Solutions Foundations of Program Design Addison Wesley* 2001
6 King K.N. *Java Programming from the Beginning* Norton 2000
7 Liang, D.Y. *Introduction to Java Programming* Prentice Hall 2003
8 Riley, D.D. *The Object of Java Introduction to Programming Using Software Engineering Principles* Addison Wesley 2002
9 Felleisen, M. Findler, R. Flatt, M. Krishnamurthi, S. *How to Design Programs An Introduction to Computing and Programming* The MIT Press 2001
10 The interested reader can download the code from http://cs.bilgi.edu.tr/~chris/akab
11 Abelson, H. Sussman, J. and Sussman, J. *Structure and Interpretation of Computer Programs* MIT Press, 1984
12 Felleisen, M. Findler, R. Flatt, M. Krishnamurthi, S. *How to Design Programs An Introduction to Computing and Programming* The MIT Press 2001
13 Felleisen, M Findler, R. Flatt, M. Krishnamurthi, S. *Structure and Interpretation of the Computer Science Curriculum* Journal of Functional Programming 14, pp 113-123, 2004
14 Kernighan, B.W. and Plauger, P.J. *The Elements of Programming Style* Yourdon 1974