The Novice Programmers' Syndrome of Design-by-Keyword

David Ginat

CS Group, Science Education Department
Tel-Aviv University, Israel
ginat@post.tau.ac.il

ABSTRACT

In the course of reading the description of a given assignment, it is natural that associations with design patterns directly tied to explicit keywords or phrases in the assignment text will evolve. However, explicit keywords may not always be the basis for the desired solution. Implicit cues may yield a better outcome. This paper presents a study of novice programmers who are misguided by explicit keyword associations. The study shows that students' tendency to "design-by-keyword" may sometime lead them to incorrect or inefficient programming solutions. The study displays student solutions to three CS1 problems, each answered in three different ways. The first two ways reflect undesirable "design-by-keyword" outcomes, and the third way encloses the desired solution, which demonstrates the importance of looking for implicit cues.

Categories & Subject Descriptors

K.3.2: Computer and Information Science Education – Computer Science Education.

General Terms: Algorithms, Performance, Human Factors.

Keywords: Program Design, Student Errors, Pedagogy.

1. INTRODUCTION

Given a programming assignment, how is one's design affected by selected keywords recognized in the text? Naturally, many of us may associate particular terms in the task description with familiar design schemes. Yet, we do not hastily decide on the programming patterns suitable for the solution. As experts in our domain, we carefully examine invoked associations, and analyze their pros and cons, before deciding on the most elegant and efficient solution. Do our students do the same? Not necessarily.

In the course of examining a group of motivated high-school students, who completed their CS1 studies, we noticed many who hastily turned to programming patterns directly associated with explicit keywords in the text of their programming assignments. This paper describes our study of this phenomenon, which we call *the design-by-keyword syndrome*.

Programming patterns are well advocated, at different levels of abstraction and complexity, in the fundamental programming courses. Research in computer science education revealed that experts organize their knowledge in conceptual design schemes that combine aspects such as cause, rules, representations, and tradeoffs [4,6,7]. In addressing the unfortunate novices' tendency to focus on language constructs, educators offered in the last two decades various methods for incorporating the explicit notion of design schemes and programming patterns.

The underlying theme in these methods is the elaboration of recurring and reusable programming constructs, which were called templates, idioms, plans, or schemas [4,6,7]. Recently, they all are considered in the category of design patterns [1,5]. These constructs serve as essential means in teaching fundamental design principles, with particular emphasis on modularity.

Teaching with emphasis on modularity and design patterns indeed assists the novice programmer in skillfully organizing programming knowledge. However, the problem solving process in programming requires more than skillful knowledge organization. The recognition of the design patterns relevant for a solution should occur only after a primary stage of problem analysis. We experts are well aware of this essential stage. Our novice students are far less conscious of its importance. This study shows that a non-negligible percentage of novices skip this stage, and rather rapidly select primary design patterns that come to mind due to explicit keywords in the programming assignment text.

The study involved a group of students who were required to solve three CS1 problems. The problems varied gradually in their level of difficulty. While we expected the students to perform mildly on the second and third problems, we were surprised to see that they already mildly performed on the first problem. The next section displays the study setup. Sections 3, 4, and 5 display the results of posing the three problems to the students. Section 6 includes a discussion of the results and their tie to similar phenomena noticed in the domain of mathematics education. The section concludes with a pedagogical suggestion for computer science educators.

2. STUDY SETUP

The study was conducted in the beginning of the year 2002 with a group of 31 motivated 12th grade students, who were in the middle of their CS2 studies. The students were preparing to compete in our national programming contest, which emphasizes the notion of program efficiency in addition to correctness and modularity.

The students were given three problems, for which they were requested to provide a hand-written code in the programming language they prefer (Pascal or C). The amount of time was sufficient – 120 minutes, plus an extension for those (very few) who asked for extra time.

ITiCSE'03, June 30–July 2, 2003, Thessaloniki, Greece.

Each of the three problems can be solved in more than two different ways, which significantly vary efficiency-wise. The naive solution for each problem can be directly derived from the problem description. The most involved solution requires capitalization on underlying characteristics that are implicit in the problem description. The underlying characteristics are all derived from simple implicit cues.

We collected data in two ways. First, by simple statistics of the various student solutions, and then from explanations of specific students, whom we interviewed on their solutions. The interviews focused on the students who provided the less efficient solutions.

The following three sections present the problems posed to students, in the order they were presented – problem-1 is displayed in section 3, problem-2 in section 4, and problem-3 in section 5. After the presentation of each problem, we display the students' diverse solutions and describe some of their explanation. In the description, we draw the attention to the students' reference to explicit keywords that appear in the text of the problem descriptions.

3. KEYWORDS VS. EXTREMAL CUES

Problems-1 involves the question of whether some pair of elements, out of a possibly long sequence of elements, satisfies a sought-after property.

> **Problem-1.** Develop a program whose input is a sequence of N sizes of boxes and its output is a message notifying whether there is a pair of boxes in which one box is at least twice the other.
>
> Your program should be as efficient as possible, time-wise and space-wise. Assume that N may be very large (e.g., 1,000,000).
>
> **Example.** For the input 17 13 15 21 the output should be "No", and for the input 18 29 34 13 the output should be "Yes".

The problem was solved in three different ways, which we call: *all-pairs*, *semi-extremal* and *extremal*. We describe the three solutions, and particularly elaborate on the first, more interesting solution with respect to the paper's topic – the design-by-keyword syndrome.

All-pairs solution. Eleven students (35%) solved the problem by examining all combinations of box pairs. They indicated that the explicit text phrase in the problem description "... whether there is a pair of boxes ..." inspired them to follow such examination. Their solution saves the input sequence in memory, generates all the combinations of box pairs and compares the elements in each pair with one another. This solution requires computation time of $O(N^2)$ and space of O(N).

We interviewed some of these students, and asked them to elaborate on the way they reached this solution. Many of them indicated that they figured out from the text of the problem description that a pair of elements with a particular property must be sought-after. In their view, the text yielded an "explicit cue" in this direction, which they followed.

Associating this explicit reference in the text with their (organized) knowledge of design patterns, they realized that they could invoke a "pair generator" pattern and solve the problem by examining all the generated pairs. They indicated that this association occurred rather rapidly, and created the feeling of correct and clear solution direction.

We asked whether they noticed that the input sequence might be very long, and not fit as a whole in memory. Some indicated that they noticed this possibility, and were not sure how to handle it. They wondered whether there might be a more elegant and efficient way to solve the problem, yet decided that their approach is "sound enough".

Semi-extremal solution. Six students (19%) provided the unexpected solution of first finding the minimum and then comparing each of the sequence elements to the minimum. These students recognized an implicit cue leading to an extremal characteristic, and capitalized on it. However, they did not follow it all the way and therefore provided a solution that reduced the computation time to O(N), but still required O(N) space.

Extremal solution. Fourteen students (46%) provided the elegant and most efficient solution of finding both the minimum and the maximum and then determining the output by comparing them to one another. These students fully followed the implicit cue of extremal characteristics and recognized the underlying min-max pattern. This solution requires only one scan of the input, in O(N) computation time and O(1) space.

Notice that the last solution is not only more efficient time-wise and space-wise, but also the only relevant solution in case the input sequence is very long (1,000,000 elements). Although the possibility of very long input is explicitly stated in the problem description, only 46% of the students solved the problem for such case.

4. KEYWORDS VS. ORDERING CUES

Problems-2 is our colorful, novel invention. It involves computation of distances between elements in a sequence.

> **Problem-2.** 2N dots are spread over a long line, such that the distance between every two adjacent dots is 1 hop. N of the dots are red, and N are blue. The dots of each color are numbered 1..N. The 2N dot numbers are randomly permuted, except for one condition – for each i, i-red is to the left of i-blue (yet, i-red may be to the right of j-blue, for any j other than i). Every i-red dot has to be connected to its paired i-blue dot. The total connection lengths should be calculated.
>
> Develop a program whose input is the description of the dot permutation, and its output is the sum of the N distances between pairs of the same-index-dots; i.e., the number of hops between 1-red and 1-blue plus the number of hops between 2-red and 2-blue, etc.
>
> Your program should be as efficient as possible, time-wise and space-wise, as the input may be very long.
>
> **Example.** For the input 4r 2r 2b 3r 4b 1r 3b 1b, the output will be 10. (The distance 2 between the two 1's, plus the distance 1 between the two 2's, etc.)

This problem was solved in three different ways, which we call: *repeated-scanning*, *locations-array*, and *ordered-colors*. We display the three solutions and elaborate on the first two.

Repeated-scanning solution. Seven students (23%) solved the problem by saving the input in an array and scanning it N times – once for every red-blue pair. The students indicated that they extracted from the text the requirement to calculate the distance between every i-red and i-blue, and therefore separately scanned the data for each i. This solution requires $O(N^2)$ time and O(N) space.

The students explained that the text in the problem description explicitly requires distances between pairs of elements, and therefore the most natural way to follow is to design the calculation accordingly. Repeated scanning – one scan for each pair of dots – was their first association, and the obvious design pattern to use. They speculated that there might be a more elegant and efficient solution, but did not come up with one. As in the previous problem, we asked them how their solution will cope with very long inputs. They realized the difficulty, but were nevertheless satisfied with "the reasonable approach of their solution".

Locations-array solution. Fifteen students (48%) solved the problem by scanning the input only once, using an array that records for each i the location of i-red. These students noticed that when the location of i-red is recorded, the distance between i-red and i-blue can be calculated upon finding the location of i-blue. Thus, in the single scan over the input, reading i-red yields recording of the location of the first element in a pair, and reading i-blue yields the addition of this pair's distance to the total sum of distances. Both time and space complexities are O(N).

These students were very fond of their one-input-scan solution. They felt that compared to repeated scanning their solution is very elegant and nicely capitalizes on the notion of 'location recording'.

We mentioned that we doubt whether their solution sufficiently capitalizes on the pair property described in the problem description (i-red to the left of i–blue). Nevertheless, they were very satisfied with the design pattern they applied, and conjectured that one cannot do better than that, since one has to somehow record the location of the first element in each pair. We questioned the latter argument, and they insisted that "one has to perform a calculation per pair" and therefore one must know the location of the first pair-element upon finding the location of the second element.

Ordered-colors solution. Nine students (29%) provided the very elegant and most efficient solution of scanning the input only once and adding the location of each input element according to its color – a red element with a minus sign and a blue element with a plus sign. These students noticed that the input characteristic that i-red always appears before i-blue implies that the contribution of a red location to its pair-distance is always negative, and the contribution of a blue location is always positive. Capitalizing on this implicit ordering cue, they determined that there is no need to separately calculate the distance for each pair, but rather accumulate the location of each dot in a negative or a positive sign, according to its color. This pattern implies a very simple solution of O(1) space.

5. KEYWORDS VS. AVERAGE CUES

Problems-3 involves the generation of elements that satisfy a particular, given property.

> **Problem-3. a.** Develop a program whose input is a positive integer N, N<100, and its output is all the <u>triples</u> of positive integers with average N. **b.** Output for the input of part-a all the <u>sets</u> of three <u>distinct</u> positive integers with average N. (Notice that there is no difference between sets like {1,2,3} and {2,1,3}, whereas the triples <1,2,3> and <2,1,3> are different, since the order of elements in a triple matters.)
>
> Your program should be as efficient as possible, time-wise and space-wise.
>
> **Example.** For the input 3, the output for part-a will include triples such as <1,1,7>, <3,4,2>, <1,7,1>, and many more, and the output for part-b will be exactly the three sets: {1,2,6}, {3,5,1}, {2,4,3}.

This problem was solved in more than three different ways, some of which were erroneous. We display the three main solution approaches – *brute-force*, *distance-perspective*, and *sum-perspective* – and elaborate on the first two.

Brute-force solution. Eleven students (35%) solved part-a of the problem by generating all positive triples from <1,1,1> to <3N,3N,3N>, checking for each generated triple whether the average of its elements is N. The students indicated that they viewed the problem requirement of triples of the same average as a condition that has to be checked for all generated triples in the relevant range (some bounded the range a bit differently from the above). They wrote three nested loops, in i, j, and k, bounded each of them by 3N, and checked the average of each generated triple <i,j,k>. Their solution requires $O(N^3)$ time.

Part-b was solved only by some of these students. Those who solved it followed one of two approaches. One approach involved the checking of element distinction in addition to average=N. Each set is output six times, rather than one, since the same three distinct integers are repeatedly generated, in different orders. A few students tried to take an approach that avoids this repetition, by marking in memory "the sets output-so-far". This implies $O(N^3)$ space, and very careful programming, which some of them did not display.

The students indicated that they figured out from the text of the problem description that the problem is a generation problem, and since a particular property has to be satisfied, each generated tuple (triple / set) should be checked for satisfying the property. As in the previous problems, in their view the text yielded an "explicit cue" - a particular property, which they addressed.

Associating the "explicit text cue" with their (organized) knowledge of design patterns, they decided that a triple-generator will embody the underlying design template. Here again, they mentioned that this association occurred rather rapidly.

We indicated to them that in their solution many tuples are generated and not displayed, and it would be much more efficient to directly generate only the necessary tuples. They agreed, but mentioned that they did not have an association with a relevant design pattern that yields the necessary selective generation.

We asked why they did not capitalize on some implicit cues, tied to properties of the average. They noted that they did not recognize helpful cues, and their experience yields successful designs by extracting the important information that can be interpreted from explicit clauses of the problem description.

Distance-perspective solution. Eight students (26%) tried to solve both parts by orderly generating one element i "on one side of the average" and the other two j, k "on the other side". These students capitalized on the property that the sum of the distances of the three elements in a triple (set) from the average is 0. They did not generate unnecessary triples. However, their focus on the "total-distance 0" property as the underlying principle for the solution design yielded many erroneous solutions. Almost all of them had some computation error. The time required for their solutions varied from the order of the output size up to $O(N^3)$.

These students noticed the average characteristic of "total-distance 0", or "average as a point of balance", and tried to capitalize on this characteristic. However, it seemed that they were caught-up with the term "average", and did not pay attention to another characteristic: "average=N is equivalent to sum=3N". This characteristic shifts the view from average to sum, and considerably simplifies the computation. Being occupied with the keyword "average", which appeared in the text of the problem description, they did not perform this perspective shift. Their association involved the generation of the relevant elements "around the average".

Sum-perspective solution. Twelve students (39%) provided the elegant solution of systematically generating three integers that sum to 3N. They realized the equivalence between average=N and sum=3N, and noticed that a design based on the latter is much simpler. For part-a, they "ran" the first triple element i from 1 to 3N-2, and generated for each value of i all the j,k pairs that sum to 3N-i. For part-b, they generated all the triples of integers in which i<j<k. Their solutions were mostly correct, and yielded optimal computation time.

6. DISCUSSION

In reflection on the various solutions of the three problems, we observe that the first one or two solutions of each problem were directly derived from keywords, phrases, or clauses, explicitly mentioned in the problem descriptions. These solutions were sometimes inappropriate for some of the inputs and less efficient than the better, optimal solutions. The better solutions were derived from implicit cues in the problem descriptions.

The statistical data show that a significant number of students designed the less desired solutions. The interviews that were held with some of these students revealed that they often followed initial associations they had upon reading the problem description. The students indicated that particular, explicit terms in the text invoked their associations, sometimes rather rapidly. These associations led them to design patterns with which they were familiar, and yielded the programming patterns (schemes) of their solutions.

When we asked these students about implicit cues and characteristics that could be inferred from the explicit problem descriptions they mentioned that they wondered about such cues, did not find any useful ones, and felt satisfied with their developed solutions. While some of them sensed that there might be a solution better than theirs, they felt that their way of directly associating design patterns with text descriptions embodies a "natural and effective approach".

Their "effective approach" led them to generate all the pair combinations in Problem-1 due to a text phrase of the type "... is there a pair that satisfies some condition ...". This approach led them to generate all the triple combinations in Problem-3 due to a text phrase of the type "... all the triples that satisfy some condition ...". It also led them to view the notion of "a pairs of elements", mentioned in the text of Problem-2, as an atomic, inseparable unit, with respect to the computation of its inner distance.

The students who did not follow this approach identified and followed cues that were implicit in the problem descriptions, and yielded characteristics and properties on which they could very elegantly capitalize.

The approach described above, of directly translating the problem description text into a solution, reflects a *design-by-keyword* syndrome that characterizes a non-negligible amount of novices. While such an approach may be the right one to follow in the solution of some programming tasks, it might also yield unsatisfying outcomes.

Unsurprisingly, this approach is not unique to computer science students. In the domain of mathematics education studies have shown that less competent students demonstrate a keyword approach in problem solving. Briars and Larkin called it "the keyword method" [2]. Hegarty et al. describe this strategy as a short-cut approach, and call it the "direct-translation strategy" [3]. They show that in the solution of word problems, less competent students combine numbers and keywords based on direct (often erroneous) translation of the problem text, as opposed to more competent students, who follow a "problem model strategy".

Although our study focused on high-school 12th grade students, we believe that the phenomenon we describe also occurs among college and university students. We conjecture that such a phenomenon is typical of students who are rather rapidly occupied with keywords or phrases in the problem description. While this occurrence may sometimes be considered natural, and yield reasonable results, students should be aware of its potentially less desired outcomes, as shown in this paper.

Computer science educators should notice their students' "design-by-keyword" syndrome and address it. While we all advocate the use of modularity and design patterns, we should realize that some unfavorable student tendencies might evolve. It is important to show the students that problem analysis is an essential stage, prior to design. Good design should capitalize on thorough analysis, which should involve the cues implicit in the problem description. The three problems presented in this paper may serve as useful examples for illustrating the significant role of implicit cues and the less desired outcomes of hasty "design-by-keyword".

REFERENCES

[1] Astrachan O., Berry G., Cox L., and Mitchener G., Design patterns: An essential component of CS curricula, *Proc of the 28th SIGCSE Technical Symposium on CS Education*, (1998), 153-160.

[2] Briars D.J. and Larkin J.H., An integrated model of skill in solving elementary word problems *Cognition and Instruction*, (1984), 245-296.

[3] Hegarty M., Mayer R., and Monk C., Comprehension of arithmetic word problems: A comparison of successful and unsuccessful problem solvers, *Journal of Educational Psychology, 87*, (1995), 18-32.

[4] Linn M.C. and Clancy M.J., The case for case studies of programming problems, *Comm of the ACM, 35*, (1992), 121-132.

[5] Linn M.C. and Clancy M.J., Patterns and pedagogy, *Proc of the 29th SIGCSE Technical Symposium on CS Education*, (1999), 37-42.

[6] Rist S.R., Schema creation in programming, *Cognitive Science, 13*, (1989), 389-414.

[7] Soloway E., Learning to program = learning to construct mechanisms and explanations, *Comm of the ACM, 29*, (1986), 850-858.

[8]