A Study of Loop Style and Abstraction in Pedagogic Practice

David J. Barnes
School of Computing, The University of Kent,
Canterbury, Kent. CT2 7NF, UK
d.j.barnes@kent.ac.uk

Dermot Shinners-Kennedy
Department of Computer Science and Information Systems, University of Limerick, Limerick, Ireland.
dermot.shinners-kennedy@ul.ie

Abstract

This paper describes the results of a study into the use of structure and abstraction in the programming styles of lecturers and teaching assistants involved in teaching programming to students attending university and other third-level institutions. The study was motivated by the hypothesis that the trend towards object-orientation is being matched by pedagogic materials that consistently foster the deployment of abstraction and structure in the solution of programming problems. Unfortunately the evidence does not support the hypothesis. We conclude that the persistent use of abstraction at all levels of implementation is necessary to perfect expertise in its application and secure the benefits of the object-oriented paradigm.

Keywords: Abstraction, exit-in-the-middle problems, iteration, object orientation, pedagogy, structured programming.

1 Introduction

We have been using programming languages for well over half a century. It has been a period of persistent change during which the evolution of programming practice has had at least three identifiable 'ages': the programming-in-the-absence-of-discipline age; the importance-of-structure-and-abstraction age; and, the adoption-of-object-orientation age. The relative brevity of these ages emphasizes the speed with which we have experienced the transition from one to the next. During this period, the programming community has succeeded in creating artefacts of great variety and immense complexity, albeit not always with the preferred levels of reliability and utility. The history of these ages maps the maturation of the discipline and the emergence and recognition of abstraction as the "key to computing" (Kramer, 2007).

Ostensibly the history of programming pedagogy has proceeded hand-in-glove with developments in programming research and practice. For example, the widespread adoption of object-oriented programming as the paradigm for introductory programming courses in the late 1990s is one indication that the academic community is often anxious to support the changing industrial landscape, parallel the tool deployment of practitioners and embrace curriculum development to offer students the greatest exposure possible to pivotal concepts like abstraction.

Honouring the research tradition that our role is to question and not to worship we sought to investigate the deployment of abstraction in the pedagogic materials and exemplars of programming teachers. Our motivation derived from the incidence of sample solutions in textbooks and other materials which were either equivocal in the use of abstraction or completely devoid of it. This appeared to lend credence to David Gries' recent summary of programming instruction which he described as a "muddled situation" and noted, "For 50 years, we have been teaching programming … And yet, teaching programming still seems to be a black art … In some sense, we are still floundering, just as we were 50 years ago." (Gries 2008)

In the following sections we provide some historical background to characterize three 'ages' of programming which led to the emergence of the centrality of abstraction; present the problem we set for participants in the study; describe the results we observed; and provide some discussion and conclusions.

2 Structure and abstraction

It seems reasonable to characterize the era of early machine-level programming, through to the earliest higher-level programming languages, as the absence-of-discipline age. Of greatest concern were the goals of squeezing code into tiny amounts of memory and saving machine cycles. It is customary to see an evolution from this primordial age to an age of the importance-of-structure-and-abstraction as having its origins in the publication of Dijkstra's famous letter (Dijkstra 1968) on the harmfulness of go to statements. One of Dijkstra's main concerns about their use was the resulting increased difficulty in bridging the intellectual gap between the static, textual representation of a program and its dynamic behaviour. In a subsequent work (Dahl, Dijkstra, and Hoare 1972) he observed, "The art of programming is the art of organizing complexity, of mastering multitude and avoiding its bastard chaos as effectively as possible." He identified *abstraction* as the key mental technique for dealing with complexity and "usefully structured" programs as the resultant objects of its application. In addition to facilitating the comprehension and control of complexity, abstraction and useful structure provide the framework for developing programs that are not just solutions to a specific problem but are members of a "family of related programs" that we can think of "as alternative programs for the same task or as similar programs for similar tasks." The fact that these insights

are nearly forty years old does not diminish their significance.

Dijkstra's views sparked an intense debate, which was dominated by arguments regarding the benefits, or otherwise, of writing programs with and without go to statements but which was, in essence, about structure and the application of abstraction. Sequential search of a list of values to locate the first instance of a given value was the canonical example used by virtually every contributor to the debate.

In terms of structure, all search algorithms have two identifiable components. The first is the repetitive component which performs the actual search, and the second is the reporting of the outcome. Abstracting the essential features of the search component yields two possible termination scenarios (1) when the search space is exhausted (i.e., the end of the list is reached) or (2) when the search requirements are satisfied (i.e., the value is located).

With respect to the chosen example (i.e., sequential search) much of the debate centred on how a successful search should be handled. One side argued that successful search should be treated as an exception implemented as an early or 'premature' exit from within the repetition using, for example, a `goto` or `break` or their equivalents. Thus Knuth (1974) noted, "in the general case it is a nuisance to avoid the `goto` statements". The counter-argument treated successful search as one possible reason for a 'normal' termination of the repetition using a compound condition in the iteration construct. For example, Wirth (1974) argued, "Often the need for an exit in the middle construct is based on a preconceived notion rather than on a real necessity, and that sometimes an even better solution is found when sticking to the fundamental constructs."

Because of these characteristics, search algorithms are commonly referred to as 'n-and-a-half', 'loop-and-a-half' or 'exit-in-the-middle' problems and might be programmed in Java, for instance, in the two alternative ways shown in Figure 1. (The use of `while` is a matter of taste; a `for` construct could be used with the same effect.) It is worth noting that the `goto`-less version preserves the separation of concerns associated with the search and reporting components. In contrast, the `goto` version offers the option of conflating the detection and reporting of a successful search into a single operation (i.e., `return i` shown as an alternative inside the `while` loop).

From a pedagogic perspective the most influential contribution to the debate on the appropriateness or otherwise of exit-in-the-middle strategies was published by Soloway, Bonar, and Ehrlich (1983). They reported on an empirical study on the cognitive fit between programming language constructs and novice programmers' preferred strategy. Among other things, they concluded that, "Students write programs correctly more often using a construct that permits them to exit from the middle of the loop." This study is frequently cited in discussions concerning pedagogic approaches to looping constructs and their application (for example, see Roberts (1995) and the novice language GRAIL (McIver and Conway 1999) whose only repetition structure is a Soloway-style loop) despite Wirth's (2006) warning that "[It] defies any regular structure, and makes structured reasoning about programs difficult if not impossible."

```
// goto-less form of loop-and-a-half
public int seqSearch(int[] values, int x)
{
   int i = 0;
   while(i < values.length &&
                 values[i] != x) {
       i++;
   }
   return i < values.length ? i : -1;
}

// goto (break) form of loop-and-a-half
public int seqSearch(int[] values, int x)
{
   int i = 0;
   while(i < values.length) {
       if(values[i] == x) {
           break;
          // Alternatively, return i;
       }
       i++;
   }
   return i < values.length ? i : -1;
   // Alternatively, return -1;
}
```

Figure 1: Typical structure of 'loop-and-a-half ' problem solutions

The transition to the adoption-of-object-orientation age places abstraction and structure at the centre of everything. Partitioning state space into objects and identifying the abstract behavioural aspects of those objects are the central tenets of the object-orientated paradigm. They support encapsulation, inheritance and ultimately generic programming and, thereby, provide the infrastructure to realise programming solutions that are, as Dijkstra urged, not solutions to a specific problem but members of a "family of related programs" that we can think of "as alternative programs for the same task or as similar programs for similar tasks."

3 An informal study of personal practice

In an effort to see how influential the earlier ages of the development of programming expertise have been, and to explore the application of structure and abstraction in the programming styles of those involved in teaching programming, we ran a small informal exercise with ten academics from several Universities. With no time limit we asked them to code solutions to a problem specified by Yuen (1994), details of which can be found in the appendix. The solutions, written in a language of their choice, were submitted electronically.

The problem involves a slight extension to the standard loop-and-a-half example of determining whether or not a particular value exists in an array. The extension involves finding whether a value occurs up to two times in the array, and reporting the number of occurrences. In addition, if the value occurs twice, it is necessary to report whether the gap between the occurrences is even or odd. The problem has the attraction of being simple enough to use in a short exercise.

In a series of papers, Yuen (1983, 1984, 1994) documented solutions to various problems based on an exit-in-the-middle strategy using go to statements, and provided an extensive rationale for his preference for that strategy over a more 'abstract' or 'structured' approach.

The temporal aspects of our problem choice are important. Yuen was writing during the period of transition from the 'programming-in-the-absence-of-discipline' age to the 'importance-of-structure-and-abstraction' age. Since then we have transitioned to the 'adoption-of-object-orientation' age. We hypothesized that the evolution of programming expertise would impact on the type of solutions we would receive. We anticipated solutions exhibiting judicious application of the principles of useful structure and abstraction. In short, we expected solutions with a markedly different style to those published originally by Yuen.

3.1 The "Anticipated" Solution Style

Figure 2 is illustrative of the type of solution we expected (omitting the output details, which are not germane to the following discussion). It is a modified version of the 'go-to-less' sequential search solution shown in Figure 1. The modifications represent a refinement of the abstraction levels required in the original problem. Termination of the while loop is still dependent on an 'and' condition. The first part remains a test for the end of the array while the second part has been altered to determine if the search value count has reached two. The loop body has been altered to give effect to the new circumstances. In addition to incrementing the array index it tests for occurrences of the required value and increments a value count accordingly.

```
public void findZeroes(int[] values)
{
   int count = 0;
   int[] pos = new int[2];
   int i = 0;

   while(i < values.length && count < 2) {
       if(values[i] == 0) {
           pos[count] = i;
           count++;
       }
       i++;
   }

   switch(count) {
       case 0 : // report no occurrences
           break;
       case 1 : // report one occurrence
           break;
       case 2 : // report two occurrences
           int gap = pos[1] – pos[0];
           // perform odd/even calc's ...
           break;
   }
}
```

Figure 2: The anticipated style of solution

To report the parity of the gap between two occurrences of the search value it is necessary to record their positions. It is possible to achieve this using two simple variables. However, applying the principles of abstraction to the problem specification we are conscious that the current problem is an instance of a family of problems intended to locate N occurrences of the search value. This explains the introduction of an array for the search value positions.

3.2 The Participants' Solutions

Participants were given a free choice over the language they used, and the solutions we received were all coded in either Java or C++. There turned out to be little consistency in any elements of the solutions: the type of control structures; the number of loops; or the variables used to collect the data for reporting purposes. The solution shown in Figure 3 is illustrative of the solutions submitted. Table 1 summarizes the various search methods among these solutions. We have excluded one solution as it coded a separate method for each case (including two almost identical methods for the odd and even gap cases) and employed a different approach for each of the no-, one- and two-zeros cases. In essence, this was an example completely devoid of any structure or abstraction, and we categorize it as an outlier.

```
public void findZeros(int[] values)
{
   int pos1 = -1;
   int pos2 = -1;

   for(int i = 0; i < values.length; i++) {
       if(values[i] == 0) {
           if(pos1 < 0) {
               pos1 = i;
           }
           else {
               pos2 = i;
               break;
           }
       }
   }

   if(pos2 >= 0) {
       // Two zeros ...
   }
   else if(pos1 >= 0) {
       // One zero ...
   }
   else {
       // No zeros ...
   }
}
```

Figure 3: A typical solution from the informal study

Search methods	#
Single for-loop, with break in the middle	3/9
for-loop, scanning the whole array but recording only first two zeros	1/9
for-loop, with loop condition testing number of zeros found	2/9
Two while-loops with loop condition testing for a zero	1/9
Two for-loops with break in the middle	1/9
Library search method to find a zero	1/9

Table 1: Loop structure in the informal study

Seven solutions used for-loops, one used a pair of while-loops, and one avoided coding the search explicitly by using a library search method. Of the for loop solutions only two used a compound condition testing for the end of the array and whether the zeros had been found yet. The other five either used a break from the middle of the loop or continued to the end of the array even if two zeros had been found (the additional zeros were ignored). Both of the non-for loop solutions used a pair of loops to search for the first and then the second zero.

The use of variables also provides insights into stylistic preferences. Table 2 describes how variables were used, both to manage the flow of control and collect data for the output requirements. Typically, solutions either recorded the positions of the two zeros (*p1p2*; see Figure 3) or a combination of the number of zeros found (*nz*) along with either the position of the first (*p1*) or the distance (*gap*) between them. One of the solutions employing two loops simply toggled a Boolean variable (*even*) for the gap while looking for the second zero.

Id	Purpose	#
nz	number of zeros found	4/9
gap	running count of non-zeros following a zero	2/9
p1	position of first zero	1/9
p1p2	two variables recording positions of the zeros	6/9
even	whether the gap is even or odd	1/9

Table 2: Use of variables in the informal study

On the basis of this informal study we made two observations:

- If we ignore the solution that used a library call, half of the solutions were coded using a break from the middle of the loop. The associated break was often 'buried' some way down in the body of the loop (as can be seen in Figure 3).
- All of the solutions solved the *specific* problem that had been set. For instance, all were written to deal with a maximum of two zeros – none sought to use a data structure to record the positions of the zeros or parameterized their methods on the number of zeros to be located. This lack of generalization is particularly noticeable in the solutions that coded separate loops or used two library calls to search for the first and second zeros.

Both observations, and the disparity between the anticipated solution and the actual solutions, caused us a degree of concern. We began to wonder whether these personal practices might also be typical of *pedagogic* practice, so we conducted a second study in order to test the waters in this respect.

4 A study of pedagogic practice

We set the same problem of searching for up to two zeros to groups of academics and postgraduate students at a University that has been teaching object-orientation at introductory level for over ten years. Postgraduates were included in the study because it is common for them to provide support with programming classes, and these students also tend to be recent graduates, reflecting current academic practice. Staff and postgraduate mailing lists were used to invite voluntary, anonymous participation in the study. All submissions and questions about the exercise were handled anonymously without at any point identifying the participants, other than whether they were a member of staff or a student.

Because we were concerned that the informality of the first study might simply have lead to solutions being offered that actually had no bearing on pedagogic practice, we made an addition to the problem specification. We asked the participants to:

> "Bear in mind that [your solution] should be the sort of thing you would be willing to show to introductory programming students as an example of a 'good' solution."

The idea was to avoid any suggestion that this exercise was just about creating a program that works. It was intended to emphasize that the solution should display some degree of pedagogic value. We indicated that this was part of a research study and that solutions might be used in publications, but did not otherwise give any further motivation. Solutions were received electronically and no time limit was imposed on their production. This allowed subjects to compile and test their solutions before submission, should they wish to do so.

Sixteen submissions were received that were amenable to analysis, written in C++, Java and Python. Eight were from postgraduate students and eight from members of staff. We will not distinguish further between these groups because there was actually little difference to be observed in the styles of their solutions. Table 3 summarises the preferred loop structures in these submissions.

Loop structure	#
for-loop, with break or return in the middle	10/16
for-loop, scanning the whole array but recording only first two zeros	4/16
for-loop, with loop condition testing number of zeros found	1/16
for-loop, recording the positions of *all* zeros	1/16

Table 3: Preferred loop structure in the second study

The preference for using a for-loop over the full range of the array is complete – in contrast to the first study, no solution used an explicit while-loop. In some respects this strong feature of both sets of solutions is not particularly remarkable. While it was common in Pascal to see a clear distinction between the use of a for-loop for a definite (pre-determined) number of iterations and a while-loop for an indefinite number, the C family of languages actually blurs the distinction, and the regular for-loop (as opposed to for-each) is often taught as simply being a syntactic variant of the while-loop. Nevertheless, there is still often quite a strong association in the mind of readers of code between a for-loop and definite iteration.

Of more significance in the second study is the almost complete avoidance of an augmented loop condition to finish the search once two zeros had been identified – just one solution. Ten solutions (60%) used an embedded break, as had four (44%) in the previous study. The four solutions in the second study that used neither an augmented condition nor a break continued scanning after

finding two zeros, which meant that they had to code around losing the position or gap information they had recorded; thus making the loop body more complicated than other solutions. Notable is that one solution used a list to record the positions of all zeros and not just the first two. In C++, Java and Python this solution is particularly easy to code using a dynamic data structure, such as Java's ArrayList, which has the useful additional benefit of keeping track of the number of occurrences found.

Table 4 documents the ways in which variables were used. There is more variety here than in the first study. Two solutions used a list (a 2-element array, in one case) rather than separate variables for the positions. In addition, there were four examples of the style we have labelled *p1-flag* (see Figure 4). This can be seen as a minimalist approach to variable usage: a negative value of the position variable indicates that no zero has been found yet; a non-zero value indicates that a single zero has been found, and this value is then used, along with the current loop variable, to compute the gap when the second zero is found.

Id	Purpose	#
nz	number of zeros found	7/16
seen	whether a zero has been found yet	3/16
gap	running count of non-zeros following a zero	6/16
p1	position of first zero	2/16
p1-flag	position of first zero, with out-of-bounds semantics	4/16
p1p2	two variables recording positions of the zeros	3/16
collection	positions of the zeros	2/16

Table 4: Use of variables in the second study

```
public String findZeros(int[] values)
{
   int pos = -1;  // position of first zero.

   for(int i = 0; i < values.length; i++) {
       if(values[i] == 0) {
           if(pos < 0) {
               pos = i;
           }
           else {
               if(((i - pos) & 1) == 0) {
                   return ... // Two - odd gap.
               }
               else {
                   return ... // Two - even gap
               }
           }
       }
   }

   if(pos < 0) {
       return ... // No zeros
   }
   return .. // One zero
}
```

Figure 4: A solution from the second study (p1-flag)

It has to be admitted that the results of this repeat of the original experiment were a considerable surprise to us. We had assumed that the characteristics we had observed in the submissions to the first study were, in effect, 'quick and dirty' solutions to a relatively simple problem. We anticipated that the addition of the 'good solution' rider would result in our seeing quite different solutions – akin to exemplars that would achieve full marks in a student assignment. On the contrary, there was little material difference between the two sets of solutions. We saw just one solution out of sixteen that we felt came close to genuinely being 'a good solution', i.e., using a loop with a single exit point and a collection to store the positions of zeros. While it could be argued that we had not made it clear enough what we were expecting, the risk of being too specific is that the results will not accurately illustrate the participant's normal practice in a teaching situation. In other words, we were not interested in whether they *could* write a good solution but whether they *would*.

5 Discussion

Despite the limited nature of this study, it appears to us that several themes emerged strongly and that may well be evidence of widespread pedagogic practice:

- While the problems have an *indefinite* character, there was an overwhelming preference for definite solutions based on using a for-loop that appeared to operate over the full array.
- There was a strong preference for exit-in-the middle solutions.
- Where an exit-in-the-middle condition was used, it was *always* a physically separate and distinct test from the test for reaching the end of the data structure.
- The solutions we saw were often closely tied to the fact that only two zeros had to be identified; for instance, using either one or two variables to record the zeros' positions, rather than a collection.
- The temptation to conflate aspects of the solution was irresistible for some participants. For example, in the first study one participant toggled a boolean variable to record the parity of the gap whilst searching the list. In the second study, as Figure 4 illustrates, embedding details of the reporting requirements in the search phase was not unusual.

Our view is that these themes are, in fact, part and parcel of a single issue – an approach to program design that focuses on the immediate details of the task at hand rather than on the properties of a more general version of the task. In this study the specific task was to locate the positions of up to two zeros in an array. However, the search aspects are clearly instances of the more general problem of finding the positions of up to *N* occurrences of a value in a collection. What we believe we are seeing here is a failure to use abstraction and useful structure in program design at the method level.

Our preferred solution to the problem presented is shown in Figure 5. Employing abstraction, the problem is partitioned into two components or methods. The first is a

customisable search mechanism, with the search value and the required number of occurrences provided as parameters of the search invocation. The features of Java's dynamic List structures are used to store the positions of whatever number of values must or can be recorded. The list is returned as the result of the search (the list could, of course, be converted to an array of int if that would be more convenient.) The second method localises the specific reporting obligations and extracts whatever reporting information is required from the list received as its parameter.

```
/**
 * Find up to max occurrences of x in values.
 * Return the positions of the occurrences.
 */
List<Integer> findValues(int[] values,
                         int x, int max)
{
   List<Integer> pos =
           new ArrayList<Integer>();
   int i = 0;

   while(i < values.length &&
                 pos.size() < max) {
       if(values[i] == x) {
           pos.add(i);
       }
       i++;
   }
   return pos;
}

void reportResults(List<Integer> pos)
{
   switch(pos.size()) {
       // Cases for each reporting option.
       ...
   }
}
```

Figure 5: A solution to all the search-for-n variations

This is a solution to the "family" of problems defined by the Yuen specification. Its strength is in its simplicity; its power in its malleability; its attraction in its treatment of specifics. The solution could equally well have been coded in any one of several programming languages, including those not categorised as object-oriented. However, this particular solution clearly illustrates a mix of fundamental procedural and object-oriented elements that are appropriate even at a relatively early stage of an introductory programming course. For instance, see the introductory textbook by Barnes and Kölling (2008) where basic iteration and dynamic collections are introduced together.

One important feature of the solutions submitted in the studies is that they all worked – they were correct solutions. This is an important outcome and it should not be overlooked or undervalued. An essential property of any programming solution is that it achieves the desired result. However, as a maturing discipline, software engineering has established and substantially documented the importance of qualities such as, maintainability, portability, adaptability, reusability, flexibility. These are all premised on the application of abstraction. Yet, as the foregoing analysis highlights, the commitment to abstraction evident in the solutions submitted was equivocal.

For example, the abstract characteristics of all search algorithms centre on the iterative deployment of a search strategy until the search space is exhausted or the required item is found – whichever occurs first. These characteristics were realized in a potpourri of styles but the dominant one separated the detection of the conditions using an exit-in-the-middle mechanism. Only one solution, submitted in response to the second exercise, implemented the iteration in a manner consistent with the abstracted characteristics of a search.

Similarly, recording the positions of the located search values was typically handled using simple variables. Only two participants in the second exercise adopted a collection approach to the recording of the positions.

Significantly, none of the solutions parameterized the search to facilitate variation of the search value or the number of occurrences required. Neither did any of the solutions partition the problem by separating the searching process from the reporting process. In fact, many of the solutions conflated the two processes and implemented them as a single embedded block.

Of course, it is always easy to offer criticism of others' programs and that is not our purpose. Our purpose is to highlight the application of abstraction. Our conclusion is that, despite all of our aspirations and justifications for the adoption of programming tools that provide extensive support for the exploitation of abstraction, we have not, as a community, developed a premeditated disposition for its application. In fact, what we have developed is an approach founded on the principle that when we really need it we will be able to recognize that need and apply it appropriately. We are operating on "just in time abstraction." That is a precarious perch to position ourselves on.

We suspect that this is (in part, at least) a result of an 'incomplete birthing' of the age of object-orientation. A noticeable, and potentially unhelpful side-effect of the arguments into the relative merits of procedural and object-oriented approaches has been that they are sometimes treated as completely distinct animals. This effect is observable, for instance, in a relatively recent catalogue of novice OO misconceptions (Sanders and Thomas 2007) where procedural elements are almost entirely omitted. Yet novices must also grapple with the basic 'procedural' elements of methods, such as managing flow of control, where misconceptions are just as likely. Indeed, as Garner, Haden, and Robins (2005) found, it was such procedural elements that often caused more problems than the object-oriented ones. It may well be that in our desire to teach good and genuine object-orientation there has been an unintended neglect of the need for good practice in those 'procedural' elements that are an almost inevitable part of any OO program.

6 Conclusion

Dijkstra (1989) once observed that, "Breaking out of bad habits, rather than acquiring new ones, is the toughest part of learning." If we are in the habit of neglecting the deployment of abstraction techniques at the lower levels of design then our ability to deploy them at the higher levels may be compromised. The acquisition of an

habitual tendency for the application of abstraction is a formative process that demands persistent nurturing and pervasive instantiation. We cannot turn it on and off.

The importance of practice is an established mantra of programming teachers. Lack of practice can hinder the development of experience and competence. Indeed, just as an athlete or an artist develops reliable technique through repetition, the act of consistently applying abstraction – whatever the problem – is what ultimately enables a programmer to employ it with greater facility. Gladwell (2008) cites the "10,000 hour rule" which asserts that "ten thousand hours of practice is required to achieve the level of mastery associated with being a world-class expert – in anything." He also notes that, "Practice isn't the thing you do once you're good. It's the thing you do that makes you good."

Lamenting a general lack of application of discipline in the field of software engineering, recently, David Parnas laid the blame firmly in the court of teachers, saying that, "We need to: teach [students] what to do and *how* to do it – even in the first course [and] use those methods ourselves in *every* example we present." (Parnas 2010)

A particular motivation for striving for understanding in problem solving is the need to manage the increasing complexity of software systems that we are now able to build. As computer science graduates leave university and go on to work on real projects with potentially massive impacts on society, those of us who teach have a responsibility to ensure that the mindset they take with them will lead to the creation of comprehensible software artefacts. In fact, this argument is really no different from those made nearly fifty years ago over how to deal with the 'software crisis' (Randall 1996) – a phrase we don't hear so often now, but which is surely just as pertinent now as all that time ago?

We believe that it is essential for teachers of introductory programming to revisit the ideas of useful structure and abstraction in order to ensure that students are taught to apply it from the ground up in designing solutions to software problems.

We conclude with the observation that practice in the current age of object-orientation may have forgotten something of the conclusions of the preceding ages and needs to revisit them in order to adequately equip students with a full set of programming skills. This is not to say that the preceding ages always lived up to the conclusions that were reached – there was no 'golden age' – but that OO practitioners need to pay just as much attention to them.

7 Acknowledgements

We would like to express our sincere appreciation to the participants in our studies, who allowed us to analyze their solutions.

8 Appendix

Yuen's zeroes problem (Yuen 1994) as we set it in both the informal and the second study:

Inspect an array of N elements to find which one of the following is true and output a message identifying the case. Arrays indexed 1..N or 0..(N-1) are equally acceptable:

a. It contains no zeros.
b. It contains only one zero.
c. It has two zeros separated by an even number of non-zeros.
d. It has two zeros separated by an odd number of non-zeros.

Clarification questions asked by the subjects in the second study included what should be returned if there were more than three zeros. The response was that the gap between the first two was the only item of interest in this case.

9 References

Barnes, David J. and Kölling, Michael (2008): *Objects first with Java - a practical introduction using BlueJ*. Pearson Education Ltd.

Dahl, O. J., Dijkstra, E. W., and Hoare, C. A., Eds. (1972): *Structured Programming*. Academic Press Ltd.

Dijkstra, E. W. (1968): Letters to the editor: go to statement considered harmful. *Communication of the ACM* **11**(3):147-148.

Dijkstra, E. (1989): On the Cruelty of Really Teaching Computer Science. *Communications of the ACM* **32**(12):1398-1404.

Garner, S., Haden, P., and Robins, A. (2005): My program is correct but it doesn't run: a preliminary investigation of novice programmers' problems. *Proceedings of the 7th Australasian Conference on Computing Education*, Newcastle, New South Wales, Australia, **106**:173-180, Australian Computer Society.

Gladwell, M. (2008): *Outliers – The Story of Success*. Little, Brown and Company: New York.

Gries, David (2008): Foreword. In *Reflections on the Teaching of Programming*. Bennedsen, J., Caspersen, M.E., and Kölling, M. (eds). Springer, 978-3-540-77933-9.

Knuth, D. E. (1974): Structured Programming with goto Statements. ACM Computing Surveys 6(4):261-301.

Kramer, J. (2007): Is abstraction the key to computing? Communications of the ACM 50(4): 36-42.

McIver, L. and Conway, D. (1999) GRAIL: A Zeroth programming language. Proceedings of the International Conference on Computing in Education (ICCE99), 43-50.

Parnas, D. L. (2010): Risks of undisciplined development. *Communications of the ACM* 53(10):25-27

Randall, B. (1996): The 1968/69 NATO Software Engineering Reports. Dagstuhl-Seminar 9635: History of Software Engineering, Schloss Dagstuhl, Germany, August 26 - 30, 1996.

Roberts, E. S. (1995): Loop exits and structured programming: reopening the debate. In *Proceedings of the Twenty-Sixth SIGCSE Technical Symposium on Computer Science Education*, Nashville, Tennessee, USA, **26**:268-272.

Sanders, K. and Thomas, L. (2007): Checklists for grading object-oriented CS1 programs: concepts and misconceptions. *SIGCSE Bulletin* **39**(3):166-170.

Soloway, E., Bonar, J., and Ehrlich, K. (1983): Cognitive strategies and looping constructs: an empirical study. *Communications of the ACM* **26**(11):853-860.

Wirth, N. (1974): On the Composition of Well-Structured Programs. *ACM Computing Surveys* **6**(4):247-259.

Wirth, N. (2006): Good Ideas, through the Looking Glass. *Computer* **39**(1):28-39.

Yuen, C. K. (1983): The programmer as navigator: a discourse on program structure. *ACM SIGPLAN Notices* **18**(9):70-78.

Yuen, C. K. (1984): Further comments on the premature loop exit problem. *ACM SIGPLAN Notices* **19**(1):93-94.

Yuen, C. K. (1994): Programming the premature loop exit: from functional to navigational. *ACM SIGPLAN Notices* **29**(3):23-27.