The Study of Programming Standards in Computer Science Programming Courses

Shmuel Rotenstreich
Dept. of EE&CS
The George Washington University
Washington, D.C. 20052

Abstract

In this report we address programming standards issues such as object naming, in-source program documentation and, to a lesser extent, the modularity issue usually referred to as program design. It is claimed that in light of the importance of programming standards, computer science students must be prepared to use standards in general, and in industry in particular. Since today's teaching practices avoid this problem, a proposal for minimal programming standards taught by computer science departments is put forward and is shown to achieve several of the goals standardization of programming practices profess to attain.

1. Introduction

A standard is a concept, technical or behavioral, chosen by a clearly defined community from a set of available, and relatively well known, concepts to be used in some community activity. The choice of the standard is not necessarily based on its preferable merits vis-a-vis the other concepts, but rather on the need, explicit or implicit, to benefit from the whole community using the same concept. A standard usually affects external features and is less prone to impact the contextual features of the activity.

The prevalence of standards in established industrial activities, and in particular, the significance of standards in the software industry, requires some reflection on the place of standards in the teaching of programming in computer science programs. Confining our view to the teaching of programming is important; the use of programming is quite intensive on both the undergraduate and graduate levels. Consequently, the relative programming skills of most students are high. Furthermore, once out of school most graduates will be involved with programming.

From the industrial viewpoint, programming has many facets that need standardization. As in any other complex and wide ranging activity, the effect of standards on industrial programming spans a wide spectrum. There are several well established concepts that can be considered standards. A good example is structured programming. Although relatively young, it was originated in the middle 70's,[3] and caught on fast. It is manifested in both programming style and the choice of high level programming languages. Other areas experience less success. Documentation standards are not industry wide, and if a standard exists at all, it is limited to the practices of a company, or even a division within a company. Program design offers something of a fad which in turn can be thought of as a standard. While in the early 80's whenever a bona fide program design method was used it was likely to be structured design,[8] the late 80's show a tendency to choose object-oriented design.[7,1] It should be noted that the two concepts are not mutually exclusive.

This report deals with three limited aspects of programming standards, object naming, in-source documentation and program modularity. We feel that the teaching of those areas as standards can contribute considerably to both students and industry. The scope of standards in large industrial program development is considerably wider than the largest yet modest student's projects. Many facets of programming standards applicable in industry are virtually non-existent in student's projects. A simple example is a requirement document. If one appears in a student's project it is extremely rare, and mostly contrived. It is, therefore, important to see our discussion in the confined university context without demanding immediate and complete impact on the industrial setting.

The rest of the report deals with those subjects. Section 2 discusses in detail the three subject matters. Section 3 suggests the programming standards for those subjects.

2. Programming Standards at the Current Time

One can think of a wide variety of programming practices that may require standardization. As oppose to the hardware community, however, the software community does not have the parallels of standard chips, resistors, rs232c, etc. Although there is some feeling that similar achievements can be made on the software side, one has not seen too many appealing proposals for a concrete standard module interface, for instance. The furtherest we got was in Parnas' information hiding.

Being unable to make major gains of the large magnitude should not discourage us from attempting to make modest improvements in less prominent areas. For instance, object naming is considered an important, although not crucial, element in the life cycle of a program. Of the same importance is the documentation one codes into the source program with comment lines.

2.1. Object Naming

Programming standards differ somewhat from company to company. They are more common in larger companies, and are less used by small companies. In other words, programming standards are essential to larger programs and are less essential to small programs.

In large programs, object naming may follow standards rather than be left to the individual programmer. Such an approach may dictate names of modules and rules to derive object names from the name of the module originating them. For instance, given a procedure named *stack*, all local variables inside it may be required to be of the form *stack_name*. Namely, the prefix of all objects defined by this procedure have to be related to the procedure name. Such a standard, which happens be to used by some companies, is quite helpful in making programs easier to follow, and it helps avoid the danger of bizarre naming practices chosen by individual programmers.

No clear standard seems to be emerging in object naming. This is not surprising since several factors work against a standard. First and foremost, it is very difficult to introduce a standard that will cover object naming across the board. Another factor that hinders a standard is the strong influence of teaching of programming on the standard; one is told that naming has to be meaningful, however, complete meaningful naming is next to impossible in very large programs. Of the other factors involved in preventing the formation of a standard,

the most important one is the expected low return from the large effort of achieving an object naming standard.

2.2. In-Source Documentation

There is a standard that mandates the use of documentation in programs. It is a standard because it is agreed upon by the community, while clearly, a program will run perfectly well without documentation, at least until the first change. In addition, it is possible to document programs in other ways. The documentation standard is abstract because it is not concrete enough to specify the details. But being a technical standard it is incomplete if its details are not spelled out. The questions here involve the detail level in the documentation, what elements have to be included, the position of those elements in the program, the relation among the elements involved, etc.

There is no standard that details the answers to those questions. There are many standardized practices employed locally by different organizations. Those practices frequently require that every module be identified by the program name and a title describing it. On top of that, one is required to add a prose section that describes the workings of the module. Each module will also include documentation detailing the global objects it refers to or changes. Some prose is normally required to accompany the code itself and the declaration of internal objects. When the module is part of a programming-in-the-large[2] scheme, the documentation may require an interface section detailing the way to connect to this module, a clear reference to external modules invoked or affected and it might require a discussion of exceptional conditions that may occur in the module.

A large program is virtually unaccessible without any documentation standards. All one has to do in order to convince himself or herself of the difficulty in understanding and maintaining large programs is to imaging such a product without any uniform appearance of in-source documentation. Such a program will be extremely confusing and even frustrating.

It appears, therefore, that although a standard does not exist, uniformity of in-source documentation is essential for any large software product. Uniformity when projected from the product level into the general appearance of software can become a standard.

2.3. Program Design

Design standards are also used by industry. One can claim that the usage of Structured Design (SD)[8] is not a standard, but rather a technique appropriately chosen for the design of programs. However, we claim that it is a standard on two counts. First, once any technique becomes widely used it also serves as a standard. Second, there are several techniques that could substitute for SD. One reason SD is adapted by many companies is that it is widely used with reportedly good results. And this type of adaptation is a step towards a standard.

To sum up our contention, as part of the programming practices there is an industry wide adherence to several, very few, design techniques that tend to render similar structural appearances to different programs. It is interesting to examine the use of these standards with respect to our overall view point.

2.4. Summary

There are two points we would like to make at this time. It was shown in this section that those programming practices we have decided to deal with do no benefit from community wide standards. It was pointed out that some of the practices are standardized within certain organizations. Despite the lack of standards it is absolutely clear that standards would have a beneficial influence on the software process.

One cannot conclude a brief discussion of programming standards without emphasizing the import role they play in overcoming some of the most stubborn complexities of programming-in-the-large. The uniformity of the products is used not only for gains in the production process, they are also an important cognitive tool throughout the software life cycle.

3. Program Standards Worth Teaching

In light of the state of standards in programming there are several question that have to be answered. First, whether there is a need to relate to standards in the teaching of computer science in the first place. Provided the answer to the question is positive, the question is what is worth teaching, and in detail, what is the best approach.

The worth of referring to standards can be demonstrated by evaluating the student's programming skills without the benefit of programming standards. Teaching programming without reference to the existence of standards clearly does not prepare the student for future work in an industrial setting. The emphasis is not on the price the student will have to pay through the adaptation to standards in an industrial framework. This is not an enormous price. The emphasis is on the fact that the absence of standards from teaching of programming causes a certain ignorance of some facets of the complexities of programming-in-the-large. At the current time, all the tools and concepts available to programmers are not sufficient to conquer the complexity of large programs. Standards, be they object naming or program documentation, are devised to help the programmer in conquering the complexity.

The next subsections deal with the three elements of programming standards we have decided to deal with. The order of the presentation was reversed in one case to fit the logical presentation of the solution before the student.

3.1. In-Source Documentation

Student best understand the problem of documentation standards. Documentation is addressed by every teacher that teaches programming or assigns programming projects. The requirements for documentation in the different course vary widely, and in many cases the number of approaches is close to the number of courses taught. This triggers two reactions, first, the student starts to believe that documentation is a bizarre artifact with no relation to the problem solved. Second, the student attempts to avoid documentation because of the reason above, and the fact that he sees no relation between his assignment and the need to document it.

It is a tradition in science to take a minimalistic approach. In our case, this will mean that the student learns about a minimal set of documentation artifacts, and this in turn becomes a standard that can be sustained and supported. In programs written in Ada, or Pascal, the basic documentation will include a module name and a short title that describes its function. An additional opening prose section following the title is needed only in some cases. If the assignment is sufficient to describe the module, no additional prose is needed. If the module happens to include implementation details that cannot be deduced from the assignment itself then an opening prose section must be added.

For example, in a class that implements a small screen editor the module whose title description is *move cursor up*, does not need added prose if moving the cursor up is done in the sequence that follows the specification of the operation. The specification, given in class or submitted by the student, plays the same role specifications play in the everyday project. In another example, if the student implements a dynamic memory scheme to support his buffering of text for the same editor, the package containing this logic has to contain enough prose to fully specify the package. The assumption in this case is that the dynamic memory scheme is implementation dependent, and as such, is not documented by the specification.

This minimalist approach claims that documentation does not have to be redundant. If the specification exist, then the module is already documented and any addition is not only costly but also a potential danger throughout the life cycle. This approach benefits from the support of a logic bound to be easily understood by students, thereby standing a better chance of implanting the standardization aspect in them. Clearly, this approach seems to us to be an appropriate standard.

In languages such as Ada one has to take advantage of the verbosity of the language and refrain from adding in-source comments that are not essential. Inheriting from structured programming the limits on module size, it is clear that many modules should be written in such a way that no additional comments have to be embedded in the code. This implies that as a rule, student are encouraged to write readable programs. This goal is not always achievable but it helps the student acquire a reasonable understanding of the role of documentation standards. An important by-product of this standard is forcing the student to write readable programs.

Considerable emphasis is put on the need to document clearly, and explicitly, all the main data structures employed. As it turns out, data structures are difficult objects to break-down without the aid of explicit comments. This effect has to do with the non-procedural nature of data structure. Students should be made aware of the need to document data structures plainly and fully. They should be made to understand that this standard is widely accepted because it is a logical consequence of the difficult of the matter.

3.2. Object Naming

Object naming is a simple problem as long as the program is small, and it turns complicated when the program becomes large. We have found it difficult to present a reasonable and natural standard before the students. The difficulty is not surprising since there seems to be similar difficulty on the industrial level. Most, if not all, the projects the student are involved with, even in group projects, are such that object naming can be meaningful. Nevertheless, the student should be aware of of naming practices that are automatically generated according to external, non-functional, rules.

A possible compromise can be achieved by making students working on relatively large programming projects adhere to an artificial standard. This standard looks as follows: object names can be meaningful. Thus, names of modules in the screen editor project might be *MoveCursorUp*, or *DisplayNextPage*. At the same time externally used objects exported by *MoveCursorUp*, for instance, will have names such as MCUcount. MCUcount in this case stands for the number lines to move up. Conflicts arising between modules with identical acronyms are to be resolved by the students. The resolution which may appear to indicate the inherent contradiction between meaningful and automatic naming should be viewed positively. It just helps the student realize the difficulties with meaningful naming.

This mixture of meaningful and standardized forms will come across well enough to achieve the goal of relating the programming standards issue to the student. It will avoid the danger of letting the student believe that object naming is unrelated to the rest of the program life cycle as well as to the development process.

Object naming standards are not restricted to meaningless names. There are plenty of standards or practices involved in meaningful naming standards. Some of the widely used ones are not necessarily correct. Most of those cases have to do with overreaction to maintenance and documentation problems of the 70's. Loop variables are a case in point. Students are frequently encouraged to name loop variables *counter* or *index* instead of *i* or *j*. However, mathematicians do not have great difficulty associating semantics with an index named *i* or *j*. We believe that students should not be encouraged to create a totally new tradition and ignore well established standards that in this case are imported from mathematics. Despite the above comments, meaningful naming is still need in certain cases of indices or pointers.

Names of other objects are also subject to the overreaction mentioned above. It is not uncommon to find variables of the shape and form of *Temporary_Queue_Pointer_2*. Supposedly, this indicates that the variable is a temporary pointer that points to a queue and is the second in a set of such variables. Usage of those extremely explicit naming standards imposes a very heavy toll. The program becomes crowded and difficult to read. The number of typos resulting from such a style is larger than the number when the object names are shorter. In addition, despite all the effort, it is still very difficult to distinguish between *Temporary_Queue_Pointer_2* and *Temporary_Queue_Pointer_1*.

Consequently, the standards that are encouraged are in line with the minimal approach. The programmer should attempt to use meaningful names for local variables, but shorthand names can still acquire meaning and are preferable to crowding the code. Thus, *Temporary_Q_Ptr_1*, or *Temp_Q_Ptr_1*, or *Tmp_Q_Ptr_1* are all better choices which, eventually, do not suffer from lose of meaning.

3.3. Program Design

As mentioned before, standardization of program design is much simpler in a university setting. Using the Ada example again, it is a common practice to talk about designing programs around the idea of Abstract Data Types [4,6] which can be represented by Ada packages. There is little point in involving students with ideas such as Structured Design, the Jackson methods[5] , etc. Student's experience is normally too limited to absorb those methods. Our practice indicates that design ideas seem too fur fetched to students especially since they are mainly concerned with coding. The same difficulty is not encountered when the idea of abstract data types is presented. Since the creation of abstract data types is very close to object-oriented design this is a good solution.

Abstract data types attain most of the advantages one needs from an external structural standard and as such is more than enough. Since this idea is well accepted in the university environment, and since it is already used by many teachers, it is already a de-facto standard.

4. Conclusion

We have suggested a minimal standard for object naming, in-source documentation and program design for programs written by students. As a minimal set, the standards are relatively easy to accept by students of all levels of computer science. At the same time, the standards are a real reflection of programming standards employed throughout industry.

We believe that understanding and accepting programming standards is important even at the level of programming study. Our scheme, therefore, guarantees an approach that compromises the needs of industry and the needs of the university and forms a foundation for the future development of programming standards and their reasonable use.

References

1. G. Booch, "Object-Oriented Design," *Ada Letters*, vol. 1, pp. 64-76, Mar, Apr 1982.
2. F. DeRemer and H. H. Kron, "Programming-In-The-Large Versus Programming-In-The-Small," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 2, pp. 80-86, Jun 1976.
3. E. W. Dijkstra, "Notes on Structured Programming," in *in O. J. Dahl, C. A. R. Hoare, E. W. Dijkstra "Structured Programming"*, Academic Press, New York, 1972.
4. J. Guttag, "Abstract Data Types and the Development of Data Structures," *CACM*, vol. 20, no. 6, pp. 396-404, June 1977.
5. M. A. Jackson, "Constructive Methods of Program Design," *Proceedings, First Conference of European Cooperation in Informatics*, vol. 44, 1976.
6. B. Liskov and S. Zilles, "Programming with Abstract Data Types," *Proceedings ACM SIGPLAN Conference on Very High Level Languages SIGPLAN Notices*, vol. 9, pp. 50-60, April 1974.
7. David L. Parnas, "On The Criteria to be Used in Decomposing Systems into Modules," *CACM*, vol. 15, no. 12, pp. 1053-1058, December 1972.
8. E. Yourdon and L. L. Constantine, *Structured Design*, Prentice - Hall, Englewood Cliffs, N. J., 1979.