Reflections on Coding Standards in Tertiary Computer Science Education

Vreda Pieterse and Derrick Kourie.

***Abstract*—Despite general agreement among practitioners and educators alike that the application of solid uniform coding guidelines is beneficial, the application of coding standards in computer science education seem to be problematic. We applied interpretive research to arrive at an understanding of the attitude of leturers towards coding standards. We identified five aspects of coding guidelines through reasoning about proposed categorisation of coding guidelines found in the literature. We discuss the viewpoints of the interviewed lecturers in terms of these aspects. We conclude with our recommendations to promote the teaching of good programming practices in higher education.**

***Index Terms*—Coding Standards, Programming Style, Teaching.**

I. Introduction

THERE is ample literature that discusses the benefits of having uniform coding styles and standards. Most readers will agree with the following without hesitation:

> *Inexperienced developers, and cowboys who do not know any better, will often fight having to follow standards. They claim they can code faster if they do it their own way. Pure hogwash. They MIGHT be able to get code out the door faster, but I doubt it. Cowboy programmers get hung up during testing when several difficult-to-find bugs crop up, and when their code needs to be enhanced it often leads to a major rewrite by them because they're the only ones who understand their code. Is this the way that you want to operate? I certainly do not.* [1]

According to Bridger and Pisano [2] coding standards are laid down to achieve robust and error free code that is easy to use, understand and maintain. Style is a crucial component of professionalism in software development. Clean code that follows stylistic conventions is easier to read, maintain, and share with colleagues. When a consistent style is used throughout a project, it makes it easier for the developers working on the project to understand each other's code [3]. Oman and Cook [4] found through empirical studies that the style used when writing or maintaining a program has a direct impact upon the quality of the software and the comprehensibility and maintainability of a program.

We need to educate students to become professional, responsible, reliable developers, capable of producing quality code. This can partially be achieved by setting a good example to the students. Ala-Mutka *et al.* [5] contends that common coding conventions are very important in software projects in practice and advocate that university courses should pay attention to these basic rules, to give a good understanding of their benefits and usage for all students. Poole and Meyer [6] contends that the requirement to adhere to solid coding standards evidently leads to the development of quality programming habits which students are able to demonstrate in producing quality code.

Adhering to styles and standards based on good programming practices are not only beneficial for sharing and understanding code among programmers. These practices can also increase the productivity of individual programmers and simplify the evaluation of code written by students. Programmers who code in good style are less likely to have silly bugs and will actually spend less time developing and debugging their code [7]. Zaidman [8] reports improved appearance and reliability of student programs after the adoption and enforcement of departmental programming style and coding guidelines at Mary Washington College in Fredericksburg.

Despite the general agreement among practitioners and educators alike that the application of solid uniform coding guidelines is beneficial, standardisation of guidelines has eluded us. Enforcement also seems to be particularly problematic in educational settings. We embarked on research to investigate the teaching of coding standards in higher education. In the following section we discuss our research design.

II. Research Design

A. Interpretive Research

Trying to reach an understanding of issues concerned with teaching and learning, implies getting to grips with a whole range of human issues such as the attitude of students, the politics within departments and the ethos and environment of the institition [9]. Understanding grows through experience, observation and contemplation. Lewins and Silver [10] reminds us that the role of the interpretive researcher is not neutral, and that the accumulation of ideas and the analysis of the data should be consciously reflexive. Interpretive studies hence seek a relativistic, albeit shared, understanding of phenomena under investigation [11].

B. Research Problem

A debate was started in the Department of Computer Science at the University of Pretoria aimed at establishing a coding standard that could be applied in the presentation of all modules in the department. Supporters of the initiative agreed with Poole and Meyer [6] who argued that when there is no uniform departmental coding convention, we do not, as a department, practice what we preach. Although there was a general consensus that it would be beneficial to lay down a standard, the lecturers in the department did not agree on the content of the document. This situation gave rise to the need to answer the following questions:

1) What are the characteristics of a coding standards document suitable for higher education?
2) What are the obstacles in teaching good programming practices?
3) How should good programming practices be taught?

C. Research Paradigm

This research was conducted within the interpretive paradigm. In the research reported on here, the aim was not to derive statistically meaningful results that would corroborate a hypothesis about the benefits of the application of coding standards in an academic setting — rather, we wanted to *understand* the attitude of lecturers towards the application of coding standards in their teaching, and to answer our research questions through arguments based on our observations. Thus, the interpretive paradigm was deemed to be appropriate for the research.

D. Credibility, reliability and ethical measures

Both authors have taught programming modules at various levels, and can therefore claim reasonable insight into the abilities and attitudes of students who are learning to program. They also have participated in conversations on the topic with other educators at various occasions. These informal discussions over a long period of time have enriched their insight and their informed opinion regarding the topics discussed in this research. The professional experience of other computer science educators/lecturers was tapped into by means of a questionnaire. In order to eliminate bias in the results, responses were analysed with the aid of an academic peer, thus ensuring inter-coder reliability. All participation was voluntary and anonymous. All participants could have ignored the invitation to answer the questionnaire without any consequences.

E. Data collection

A questionnaire containing of the following four open-ended questions was compiled:

1) Think back to the last time you awarded marks for adhering to prescribed coding standards in an assignment. What were the prescribed standards? What % weight was awarded for adherence to coding standards in relation to the complete assignment?
2) Think about the last time that you did not fully agree with a coding example in the textbook used by the students. What did you do, and why?
3) Describe the context of the last time in which you talked about readable code to students.
4) Describe the context of the last time in which you talked about elegant coding solutions to students.

These questions were asked in a manner to encourage rich qualitative responses that could lead to more insight into the attitude of the respondents. We were also interested if the respondents had specific preferences with regard to loop structures and posed the following problem to them:

5) Reflect on the following two solutions for a problem. State which one you would personally prefer, and why?

```
====SOLUTION 1============
while (true)
{
    int task = bag.in();
    if (task < 0)
    {
        bag.out(task);
        break;
    }
    //compute result
    bag.out(result);
}

====SOLUTION 2============
int task = bag.in();
while (task >= 0)
{
    //compute result
     bag.out(result);
     task = bag.in();
}
bag.out(task);
```

The respondents were also encouraged to add additional comments by concluding the questionnaire with the following prompt:

6 Enter anything else you would like to share with us regarding the teaching of coding standards.

An online survey containing this questionnaire was hosted on the server of the Department of Computer Science, University of Pretoria. Participants were invited via e-mail to complete the questionnaire. All members on the e-mail list of the Southern-African Lecturers Association (SACLA) were invited to participate. This list has 235 members. A total of 17 participants completed the questionnaire.

III. Method of Data Analysis and Interpretation

We decided on a categrisation of coding standards prior to our data analysis. Thereafter we analysed the data in phases, one category at a time. We gained insight in the attitude of lecturers towards the teaching of each of these categories, by reading their answers to all the questions searching for snippets in their answers that said something about the specific category at hand. In the following section we discuss our classification of coding standards.

IV. Classification of Coding Standards

A coding standard can broadly be defined as a set of programming styles and practices to which a group of people adhere, in the belief that such adherence contributes the overall effectiveness in producing high quality code that is understandable and maintainable. Oman and Cook [12] developed a taxonomy of styles with four major categories which they labelled general practices, typographic style, control structure style, and information structure style. According to Cobb [13] the principal elements of good programming style are the requirements of aesthetics, maintenance, and portability.

TABLE I
CLASSIFICATION OF CODING STANDARDS

	Oman and Cook [12]	Cobb [13]	Ala-Mutka *et al.* [5]
1	Typographic Style	Aesthetics	Typography
2	General Practices	Maintenance	Clarity and simplicity
3	Control Structure Style		Modularity Reliability
4		Portability	Independence
5	Information Structure Style		Effectiveness

Ala-Mutka *et al.* [5] identified six categories for the classification of elements of good programming practices namely modularity, typography, clarity and simplicity, independence, effectiveness, and reliability. For our analysis we decided on a classification that incorporated aspects of all three these classifications as shown in Table 1.

We call our first category is **typography**. It corresponds with Oman and Cook's typographic style category, Cobb's category of aesthetics as well as Ala-Mutka *et al.*'s category of typography. It deals with layout issues.

Our second category is called textbfclarity. It is about measures, besides typographic rules, that can enhance the readability and understandability of code. Some of Oman and Cook's General Practices can be seen to fall in this category, for example *"Don't comment bad code - rewrite it."*. It also includes aspects of Cobb's category of maintenance.

Our third category called **reliability** concerns the production of robust and error-free code. Most of Oman and Cook's Control Structure Style which pertains to the choice and use of control flow constructs, falls into this category. It also relates to Cobb's maintenance category. We collapsed Ala-Mutka *et al.*'s modularity and reliability categories into this category.

Our fourth category is called **flexibility**. It looks at methods to build adaptable code that is portable and can easily be changed and re-used. Oman and Cook would have included this category in their General Practices Style as one of their General Practice style examples is *"Use only ANSI standard features"*. It encapsulates Cobb's portability as well as Ala-Mutka *et al.*'s independence categories.

Our final category is **effectiveness**. It regards finding elegant and efficient solutions. We deem Oman and Cook's Information Structure Style that refers to the choice and use of data structure and data flow techniques to be included in this category. It also corresponds with Ala-Muta *at al.*'s 'effectiveness' category.

In the following section we discuss views on each of these categories. We combine ideas found in scholarly publications with the views of our participants to arrive at a recommendation regarding the characteristics of a general coding standards guideline for higher education with respect to each category.

V. VIEWS

A. Typography

The purpose of typography is mainly to improve consistency and neatness in the appearance of the code. It enhances program readability. Typographic style factors include overall program formatting, module separation conventions, identifier-naming conventions, and conventions for special-case font or type styles, statement formatting, indentation, embedded spacing and use of blank lines [4]. Naming conventions sometimes include specific prescriptions such as the use of prefixes, postfixes, upper and lower case, underscores and lexical clues in variable names to convey the type of a named entity. In object-oriented contexts, a further prescription is common — i.e. of naming classes as nouns and methods as verbs.

Rules about commenting practices in terms of the style, volume and frequency of comments are also deemed typographic.

The prominent element of typographic specification relates to indentation rules. The purpose of indentation is to reveal the subordinate nature of blocks of code. Plum and Weinberg [14] has summarized the fundamental rule for indentation which underlies almost all popular formats:

> *Each line which is part of the body of a C control structure (if, while, dowhile, for, switch) is indented one tab stop from the margin of its controlling line. The same rule applies to function, struct, or union definitions, and aggregate initializers.*

Typographic aspects of coding standards can be clearly defined. These aspects are an inherent part of coding standards. Most coding standards documents devote a considerable portion of the write-up to stylistic issues. Of the respondents who elaborated on specific standards, all but two referred to typographic aspects. Some individuals seem to equate coding standards with typographic rules. For example, two respondents gave the following answers in response to a request for specifics about the coding standards they prescribe for their students:

> *Use PLUM indentation.*

and

> *For coding in Java:*
> 1) *use capitalized whole words to compose names for classes*
> 2) *same for other names, but do not capitalize first word*
> 3) *use uppercase for names denoting final (constant) values*
> 4) *Either get NetBeans to format code or adhere to a classic C style*

Because typographical requirements are usually aimed at visualising language concepts and structures that are recognised by language compilers, much of it can easily be automated. Different styles are advocated and supported by style formatters such as NetBeans. Oman and Cook [12] contend that many style formatters and texts on style are not based on theoretic foundations or empirical evidence but are unsubstantiated subjective recommendations. Nevertheless, most developers don't need empirical proof for the application of a specific style if it feels natural to apply and if the benefits seem obvious enough to warrant its application.

When a specific standard is outlined, the typographic details should be stated and enforced. However, when a general

guideline for education is to be specified, we are of opinion that detailed typographic issues should not be specified, because it is likely that people will disagree on many of these. General specifications such as consistency and enhancement of readability by means of specified use of white space should be adequate. Individuals can be advised to supply their own interpretation of the general guidelines and apply it to the detailed level of their own choice.

B. Clarity

We distinguish between typographic (mechanical) enhancement of the readability of code, and measures that can be applied by humans, using human intelligence, to enhance the readability of code. The latter we call clarity measures. Clarity emphasises the fact that code should mainly be written for human readers to understand and appreciate.

According to Zaidman [8] program readability is not only affected by consistency of coding style but also by documentation, choice of type and identifier names, and complexity of algorithms. These issues should be addressed by appropriate clarity guidelines. Clarity goes beyond typographic rules. For example, it concerns the content of comments and the meaning of variable names, and issues such as the cohesion in modules, coupling between modules, and the size of modules.

The organisation of code also plays an important role in the clarity of the code.

Naming conventions usually specify the rules related to selecting identifier names and formatting them. In existing coding standards, the emphasis seems to be on formatting conventions, which we deem typographic. Occasionally terms like meaningful, descriptive and self-documenting are used to specify how appropriate identifier names should be selected. Deissenboeck and Pizka [15] conducted analysis of identifier names in a number of real life software systems from which it is evident that identifier naming remains a problem, due to the use of different names for the same concept, the use of names that are not as descriptive as they seem, and the use of names that are too general. As Plum and Weinberg [14] noted, standards for choosing names should serve the convenience of readers, and should not serve as a shortcut for the writer.

When it is expected that comments are included in code, guidelines often offer only information about the syntax and position for commenting. According to Nurvitadhi *et al.* [16] the textbooks they investigated offered limited guidance about what should be included in these comments. In order to raise commenting guidelines above plain syntax expectations, students should be given clear and detailed guidelines for commenting [16]. These guidelines should not only differentiate different types of comments and their purpose, but should also emphasise their content and proper writing style.

Oman and Cook [12] define good style as creating the most understandable expression of the algorithm being coded. Practical guidelines to increase comprehension sometimes include restrictions on the length of modules. Hortsmann [17] suggests that the number of statements in a member function should be limited to a maximum of 30. Plum and Weinberg [14] advises developers to set a conscious limit of 25 uncluttered statements per module to prevent modules to become difficult to comprehend.

Most of the respondents were fairly passionate about readability of code. The following answer from one respondent illustrates this:

> *One important point I emphasize is that the computer wants raw byte code: programs are written for other humans to understand. So the concept of readability is introduced early and discussed throughout the course.*

However, from their answers it is not always clear if they think about readability deeper than in typographical terms. Some answers give the impression that readability only concerns correct indentation:

> *I talk about readability when explaining the use of if statements, loops etc, and encouraging them to indent the code.*

The following answer also shows that typographical issues are dominant. However, the last remark suggests a possible concern about other issues:

> *Readable code is addressed in just about every lecture. Not only are the program structures discussed, but also how the structures should be typed to increase readability. I also discuss what techniques should be avoided.*

There is a need to convey to students (and possibly some lecturers as well) that writing a program which is *clear and readable* requires effort and creativity. We have to emphasise that code should be written in a way that humans can easily understand, even if the underlying algorithm is complicated. Aiming to facilitate such human understandability of code requires thought and energy that goes beyond simply getting the algorithm to work correctly. For example, effort should be put into simplifying the flow of logic, the selection of names as well as to the wording of comments. Poole and Meyer [6] remarked that the guidelines of good writing and the guidelines of good programming have a great deal in common.

Guidelines for the enhancement of clarity of code are not easy to specify. Some organisational rules, such as the order in which entities should be listed and how they should be grouped, are helpful but often language specific. Metrics to reduce complexity, such as limitations levels of nesting and length of blocks, are useful but regularly more creativity is needed to write code that can easily be followed. There are no rules that can guarantee an identifier name to be meaningful and descriptive. Writing concise explanations is often trickier than writing code. Clarity issues can thus not be easily addressed in a general guideline. Educators can be advised to be aware of unclear student solutions. Educators should guide students to improve by giving them constructive feedback on specific mistakes they have made.

C. Flexibility

Software is not static. Almost all software is modified continuously during its life to add features, to expand capacities, to implement new capabilities or to support different equipment.

For this reason every developer should realise that change is inevitable.

The Agile Movement announced itself through the Agile Software Development Manifesto, which was published by a group of software practitioners and consultants in 2001 [18]. The need for flexible, adaptable and agile software development has become important in the fast moving modern world. Practices such as modularity, encapsulation, independence and avoiding numerical literals contribute to flexibility.

Modular design is fundamental in development of flexible code. Visibility and information hiding conventions—such as rules regarding what should be declared as public, protected, or private — underlies modular design and contributes largely to the ease of maintenance of code. The concept of adaptability is not new. The implementation of modularity to achieve flexibility has been advocated for many years. As early as 1972 Parnas [19] also advocated modular design with information hiding in mind in a 1972 publication.

The object oriented programming paradigm, inherent in almost all programming and design courses, is geared towards the development of flexible and adaptable systems. The responses of the participants supported the notion that these concepts are seriously taught. For example one respondent said:

> *I try to show the differences between code that adheres to principles like encapsulation and code that does not.*

Another respondent emphasised:

> *There are many established good practices that I impart to the students, and assess. These are not personal preferences, by the way. But they tend to be semantic oriented, rather than syntactic. There is so much going on with inheritance, overloading, accessibility, polymorphism and it needs to be covered somewhere.*

However, some of the guidelines for flexibility may be sometimes overdone and are not equally appreciated by everybody. One respondent remarked:

> *Just last week I came across a textbook examples that declared so many unnecessary (my view!) named constants, that if made the programme very difficult to understand. In the "rewritten" example given to the students, I changed some of these named constants to literal values (where deemed appropriate).*

We suspect that most educators will be comfortable with guidelines stipulating that the basic object oriented principles should be taught and be adhered to. A guideline document that can be used in a specific situation should state the requirement, illustrate the reasons for the application of such a requirement and give some examples and counter examples to clarify what is meant by the requirement and how strict it should be applied. Unfortunately the situation and level of application of these principles is very dependent on the objectives of a module and level at which it is presented. General guidelines can at best state the requirements general terms. Educators can be advised to supply their own motivations and examples to give substance to the requirements.

D. Reliability

Everything that can be done to make code less error prone falls into this category. Code that is written to be flexible is normally also more reliable simply because the principles to enhance flexibility also contribute to localisation and easier isolation of possible errors, which in turn renders the code less error prone. When code is written in discrete modules having high cohesion and low coupling, the occurrence of unexpected errors is lower [20]. We maintain that reliability is mainly achieved by adhering to flexible guidelines. However, there are some implementation practices that are not directly related to flexibility but that are likely to reduce the occurrence of programming errors, for example having pure accessors and mutators, and declaring explicit constructors and assignment operators in C++.

According to Zaidman [8] properly written and enforced guidelines can support educators in the "tedious job" of discouraging students from engaging in self-destructive coding practices. Some simple stylistic habits can help to avoid some common bugs. One example is Ambler's [1] recommendation to write the constant on the left side of comparisons. For example write `if (7 == a)` in stead of `if (a == 7)`, to avoid the possibility of using assignment in stead of comparison. If this habit is applied, the compiler will detect the error. Although this is a valuable suggestion, it is interesting to note that in the very same document in which the suggestion is made, Ambler neglects to apply it uniformly over all the coding examples, showing that adherence to standards requires conscious effort and application.

To avoid logical errors, developers should be conscious of control flow. Horstmann [17] advise against the use of break and continue statements. He also warns against catching exceptions without handling them. They should rather be thrown to callers to handle it. If the exception does occur, silently ignoring the exception can create incorrect anomalous behaviour that could be very hard to track down [3]. Hortsmann [17] advocate the inclusion of an `else` section for each if statement in nested if statements, even if it requires the inclusion of empty blocks to clarify the control flow. These style rules may help prevent certain kinds of bugs. However, violations of those style guidelines are not particularly likely to be bugs [3].

In general respondents did not agree about many of the suggested guidelines to avoid "hard to find" bugs. The answers that respondents offered to the question relating to solutions which applied different loop structures, illustrated quite clearly that it will be hard to specify standards in this regard that would be accepted easily.

Some felt strongly for the second solution and motivated that endless loops and the use of breaks are considered harmful, For example:

> *Solution 2 is better - No break (generally bad style, although can be justified in certain contexts). It also avoid a messy "while true" condition.*

Others were indifferent:

> *Both are equally valid. No preference.*

Yet others had an opinion that none of the solutions are as good as their own preferred solutions:

> *I would recommend neither. Sorry. In a modern language, I would use a full-scale iterative control structure as in:*

```
int task;
for(task=bag.in();task<0;task=bag.in()){
  // compute result
  bag.out(result);
}
bag.out(task);
```

> *Perfect!!*

and

> *Both pieces of code are horrifying. What guarantees that 'task' becomes less than 0? 'task' is not needed anyhow. Assuming that bag.in() may at some stage yield a negative value, the second "solution" is preferable to the first. How about rather coding:*

```
while (bag.in() >= 0)
        bag.out(result);
bag.out(task);
```

In this solution the respondent created more concise code by eliminating 'task' in the loop. Yet it is used (uninitialised) after the loop is completed. It also neglects the need to compute the value of 'result' that is used by `bag.out` in the loop.

For more complicated problems one is likely to encounter even more different viewpoints about what solutions are bad and what is good.

Besides being an instrument that can be used across modules to instil good practices, the purpose of a departmental coding standards document is to introduce students to the very notion of standards and adherence thereto, as a prominent part of the daily life of many coders in industry. Without general "buy in" from staff, such a document will loose its power. It is therefore important to avoid guidelines that may not be fully supported by all staff members. It may be wise to steer clear of specific guidelines regarding ways to write more robust code and to allow lecturers to teach these aspects as they deem appropriate. Most students will gain by being exposed to a broader spectrum of techniques to avoid common errors and will be able to apply those that work for them.

E. Effectiveness

Effectiveness relates to finding elegant and efficient solutions. Very few existing coding standards touch on this topic. Cobb [13] remarks that coding standards include style and elegance, but that elegant code will have a short life if it is not maintainable. Ambler [1] also advises that optimisation is not as important as customer needs and correctness. Rising [21] allocates marks to efficiency but says that efficiency is generally not an overwhelmingly important item. Roth [22] mentions that some of the emphasis on a good algorithm design can be lost in the emphasis on style and documentation.

In an educational setting, elegancy and efficiency is important and should receive adequate attention. We will do our students a disservice if effectiveness as a category of coding standards is overshadowed by the other categories. Lecturers should explicitly teach elegance and expect students to appreciate it. Teaching the subtleties of finding clever and yet simple solutions are mostly achieved by setting a good example to the students. The following answers by respondents are illustrative of how they use examples in their teaching. Lecturers provide their own examples:

> *Elegant coding is introduced through the provision of elegant solutions to problems discussed during the lecture.*

They discuss deficiencies in student solutions:

> *I usually talk about elegant coding solutions at the first lecture after each lab. In the lab I'll see solutions that are poorly written and inefficient. I will then proceed to discuss the merits/demerits of some of the solutions and present a solution that is better and most importantly - how it was arrived at.*

They also compare different solutions:

> *I don't usually talk about elegance, but try to show how to improve code to be more efficient, readable and try to show the differences between code that adheres to principles like encapsulation and code that does not.*

In our opinion it is not possible to write general guidelines for creating more effective code. Guidelines to avoid gross inefficiency like duplication of code and unnecessary variables can be specified, but is so obvious that hardly needs to be specified in a guideline. Efficiency and effectiveness are mostly algorithmic based and inherently situational. These issues can thus not be addressed in a general guideline and can best be taught by comparison of different solutions for specific problems.

F. Summary

The above categorisation of coding standards and our analysis of the viewpoints about these issues gave us insight into their possible role in teaching good programming practices. The five categories of coding standard guidelines discussed here are intertwined and reliant on one another.

Typography is mechanical. Its application is easy and can even be automated. Its benefit in education is mainly in enhancing readability and consequently making code easier to assess. Because detailed prescriptions are controversial a general guideline should delegate responsibility for its specification to individual educators.

Clarity requires higher cognitive skills. Decisions about the clarity of code is situational and highly subjective and therefore difficult to teach and problematical to specify. A general guideline can at best only advise that educators be reactive to unclear code.

Reliability mainly relies on experience. Learning how to avoid errors is best done by having suffered the consequences of making them. Unfortunately students cannot be granted

enough experience in the limited time available to teach them. We therefore have to expect them to follow prescriptions based on our experience as well as on experience reported elsewhere. Due to its controversial nature, the specification of guidelines on reliability should again be delegated to individual educators. They should be urged to emphasise the rationale for the prescriptions the give.

Flexibility is probably the most valuable in industry and many of the industry standards that are aimed at enhancing the maintainability and agility of systems are already included in our curricula. General guidelines should state the requirements in general terms and require educators to supply examples to give substance to the requirements.

Effectiveness is a category that is highly valued in some academic circles but ignored in many industry standards. Efficiency and elegance are often achieved at the cost of clarity, flexibility and reliability. We are faced with the challenge to promote effectiveness without compromising the other valuable qualities of good code. General specifications for effectiveness are either too trivial or too specific to be included in a general guideline.

VI. Conclusion

To conclude we give our verdict on the research questions:

A. What are the characteristics of a coding standards document suitable for higher education?

We are of opinion that if a coding standards document is to be compiled, it should avoid being fastidious about superficial issues. Style guidelines like placement of curly brackets serve no purpose. Coding standards should give guidance that could ultimately instil coding habits that enhance the overall quality of code. Ala-Mutka *et al.* [5] reminds us that to be aware that, no matter which coding rules students learn to apply at the university, during working life they may be required to follow different kinds of company style guides or coding conventions. One respondent also remarked:

> *The standards that graduates are forced to learn at University probably have little or no bearing on what will be force fed in a programming sweatshop where whim and fashion sometimes prevail over common sense.*

In our view it is not possible to compile a complete generic guideline for creating quality code, mostly because quality is both subjective and situational. A general guideline that can be used as a teaching instrument across modules can at most advise individual educators to specify their own rules with regard to each of the categories. A category like typography that lends itself to clear specification can not be finalised because its detail may be controversial. The same applies to some aspects of flexibility and reliability. Other categories like clarity and effectiveness are more situational and achieving them requires higher cognitive skills. Constructive guidelines are hard to formulate for these categories. They are better taught by giving reactive guidelines when counter-examples are encountered.

B. What are the obstacles in teaching good programming practices?

Educating students to adhere to good coding practices and to embrace the values of coding standards remains challenging. It is much more difficult to teach values and norms than it is to impart technical knowledge. Furthermore most students are under the impression that our sole objective is to teach *facts* rather than *insight*, let alone *values and norms*.

A difficult objective in this regard is to change the attitude of many of the students from the frivolous view that correctness is the most important, if not the only, criteria for a good solution. Unfortunately this viewpoint is widely presumed. Rising [21] reports incidents of evaluation where correctness is the only criterion for a program grade. One respondent said the following in reference to the student's attitude:

> *making things work is seen as more important than making it "look good" - probably rightly!*

The concluding remark reveals that this lecturer partially shares the student's opinion. We regard it as our task to overturn this attitude—a task that is extremely difficult because the contrary attitude is so broadly reinforced.

We see it as our educational responsibility to bring about the general understanding that readability, neatness, clarity, maintainability, elegance, etc. are important criteria for the evaluation of programs. To value correctness alone is very narrow since most software must be read and modified and is, at best, only temporarily correct.

Another hindrance educators have to face is that commercial code is seldom refactored to perfection. As a result, such code is likely to contain many examples of poor coding style and practices. Drozdz [23], for example, analysed the various versions of Sun's JDK from a maintainability point of view. He reports thousands of so-called "code smells" — i.e. code segments where design could be improved. (Simple examples would be the occurrence of variables that are declared but never used; or the occurrence of excessively long methods; etc.) Luethi [24] is also of opinion that the Linux kernel, which is generally seen as a high quality product, contains some excellent examples of bad coding. Students are inclined to reason that it is good enough if their code matches real code written by real people in the real world. In the presence of an overwhelming body of bad examples, even in textbooks [25], it can be daunting to convince students otherwise.

C. How should good programming practices be taught?

Good educators realise that students mostly learn by example. Students will not do what we say. They will rather mimic what we do, even if we draw their attention to it not to do so. For this reason it is very important that the examples we give them adhere to everything that we believe in, to the finest detail at all times. One bad instance can undo the achievement of a large amount of good examples. Ideally the values we would like to instil should be uniformly and consistently shown to them by all the lecturers involved in the presentation of all their courses. According to Zaidman [8] programming style and coding guidelines need to be applied consistently department-wide so that students do not need to

make adjustments from course to course. Rising [21] agrees by stating that having departmental guidelines saves a lot of un-learning and re-learning when students take a course from a different instructor. However, this is much more easily said than done. Not all educators are equally devoted. Every individual educator has his or her own style of teaching and set of preferences and values. When educators are expected to alter their preferences it is possible that they can loose their enthusiasm. Consequently they may become less spontaneous and less able to influence the students. The balance between academic freedom and ensuring quality education is delicate.

If the examples to which the students are exposed are solid, the need for having guidelines for students is diminished. Zaidman [8] observed that good examples serve the same purpose as guidelines:

> *Fortunately, students tend to mimic sample programs from their textbooks and class discussions. Therefore, students in beginning classes will write sound code without paying much attention to specific guidelines*

Example driven teaching seems to be a good alternative to having specific guidelines. The total learning experience can however be enhanced if the application of guidelines is combined with a strategy of teaching by example. Roth [22] reports success when he applied the following tactics to teach students that good coding is important:

a) Showing sufficient examples of poorly written and numerous (45 to 50) examples of well written code in the course of the program,
b) requiring a rigid standard for the programs submitted,
c) expecting students to modify someone else's program using only the internal documentation provided.

In the latter assignment he matches strong and less competent pairs to swop programs and contend that the stronger students can salvage the poor writing of the less competent students, while the poorer performers can learn from the better examples given to them.

Oman and Cook [12] also suggested that the importance of writing readable programs can be learned by performing a maintenance task on a poorly written program. This should convince them that poor style severely hinders program maintenance.

Having solid guidelines and enforcing them is considered beneficial. However, Zaidman [8] found that both students and faculty agree that just following guidelines will not of itself make someone a better programmer. The ideal would be to give explicit good guidelines to the educators how to *teach* students to create quality code. Unfortunately this is even more difficult to achieve than writing a guideline for writing quality code. The best we can do is to inspire educators to appreciate the values and norms underlying coding standards and to pass the inspiration on to their students. To put it in Plum and Weinberg's words:

> *".. if we can make them program self-consciously, we shall have succeeded as teachers."* [14]

REFERENCES

[1] S. W. Ambler, "Writing robust java code. the ambysoft inc. coding standards for java v17.01d," January 15, 2000, [Online; accessed 2008-03-29]. [Online]. Available: http://www.ambysoft.com/downloads/javaCodingStandards.pdf

[2] A. Bridger and J. Pisano, "C++ coding standards," February 28, 2001, [Online; accessed 2008-03-25]. [Online]. Available: http://0-alma.nrao.edu.innopac.up.ac.za/development/computing/docs/join%t/0010/2001-02-28.pdf

[3] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *SIGPLAN Not.*, vol. 39, no. 12, pp. 92–106, 2004.

[4] P. W. Oman and C. R. Cook, "Typographic style is more than cosmetic," *Commun. ACM*, vol. 33, no. 5, pp. 506–520, 1990.

[5] K. Ala-Mutka, T. Uimonen, and H.-M. Jrvinen, "Supporting students in c++ programming courses with automatic program style assessment," *Journal of Information Technology Education*, vol. 3, 2004.

[6] B. J. Poole and T. S. Meyer, "Implementing a set of guidelines for cs majors in the production of program code," *SIGCSE Bull.*, vol. 28, no. 2, pp. 43–48, 1996.

[7] M. Litvin and G. Litvin, *Java Methods A and Ab: Object-oriented Programming and Data Structures*. Andover, Massachusetts: Skylight Publishing, 2006.

[8] M. Zaidman, "Teaching defensive programming in java," *J. Comput. Small Coll.*, vol. 19, no. 3, pp. 33–43, 2004.

[9] D. Kember, *Action learning and action research: improving the quality of teaching and learning*. London: Kogan Page, 2000.

[10] A. Lewins and C. Silver, *Using software in qualitative research : a step-by-step guide*. Los Angeles ; London : SAGE, 2007.

[11] E. Henning, W. vanRensburg, and B. Smit, *Finding your way in qualitative research*. Pretoria : Van Schaik, 2004.

[12] P. W. Oman and C. R. Cook, "A taxonomy for programming style," in *CSC '90: Proceedings of the 1990 ACM annual conference on Cooperation*. New York, NY, USA: ACM, 1990, pp. 244–250.

[13] L. Cobb. (2004, 10) C and c++ source code analysis using codecheck. Abraxas Software, Inc. [Online; accessed 2008-04-11]. [Online]. Available: http://www.abraxas-software.com/pdf/ccuser.pdf

[14] T. W. S. Plum and G. M. Weinberg, "Teaching structured programming attitudes, even in apl, by example," in *SIGCSE '74: Proceedings of the fourth SIGCSE technical symposium on Computer science education*. New York, NY, USA: ACM, 1974, pp. 133–143.

[15] F. Deissenboeck and M. Pizka, "Concise and consistent naming," *Software Quality Journal*, vol. 14, no. 3, pp. 261–282, September, 2006.

[16] E. Nurvitadhi, W. W. Leung, and C. Cook, "Do class comments aid java program understanding?" *Frontiers in Education, 2003. FIE 2003. 33rd Annual*, vol. 1, pp. T3C–13–T3C–17 Vol.1, 5-8 Nov. 2003.

[17] C. S. Horstmann, *Computing concepts with Java essentials*, 3rd ed. Hoboken, NJ: Wiley, 2003.

[18] A. Cockburn, *Agile Software Development*. Boston: Addison-Wesley, 2002.

[19] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.

[20] C. Y. Baldwin and K. B. Clark, *Design Rules, Volume 1, The Power of Modularity*. Cambridge MA: MIT Press, 2000.

[21] L. Rising, "Teaching documentation and style in pascal," *SIGCSE Bull.*, vol. 19, no. 3, pp. 8–9, 1987.

[22] R. W. Roth, "The teaching of documentation and good programming style in a liberal arts computer science program," *SIGCSE Bull.*, vol. 12, no. 1, pp. 139–153, 1980.

[23] M. Z. Drozdz, "A critical analysis of two refactoring tools," Master's thesis, Univeristy of Pretoria, November 2007, supervisor: Derrick Kourie and Andrew Boake.

[24] R. Luethi, "Position paper for the xp-2003 workshop: Making free/open-source software (f/oss) work better," in *Proceedings of Workshop at XP2003 Conference*, B. Fitzgerald and D. L. Parnas, Eds., Genoa, 28 May 2003, pp. 28–29.

[25] K. Malan and K. Halland, "Examples that can do harm in learning programming," in *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. New York, NY, USA: ACM, 2004, pp. 83–87.