Supporting Students in C++ Programming Courses with Automatic Program Style Assessment

Kirsti Ala-Mutka
Tampere University of Technology, Tampere, Finland
kirsti.ala-mutka@tut.fi

Toni Uimonen
Solita, Tampere, Finland
toni.uimonen@solita.fi

Hannu-Matti Järvinen
Tampere University of Technology, Tampere, Finland
hannu-matti.jarvinen@tut.fi

Executive Summary

Professional programmers need common coding conventions to assure co-operation and a degree of quality of the software. Novice programmers, however, easily forget issues of programming style in their programming coursework. In particular with large classes, students may pass several courses without learning elements of programming style. This is often due to shortage of tutor work to give students thorough feedback on their coursework. Incorporating issues of style into programming courses is too often neglected and students are hoped to learn these issues by themselves.

To deal with the problem, a set of coding rules was collected and justified to be used in C++ programming courses of the university. An automatic C++ programming style analyzer tool was implemented to ensure that students were following the rules. Students can freely use this tool to improve the quality of their coursework, and tutors can use it for assessing the assignments. Assessment rules and criteria can be easily adjusted according to the needs of the course or the exercise in question. Since basic programming style issues are assessed by the students independently before coursework submission, the teaching staff can concentrate on giving feedback on the more advanced features of program design and course specific issues.

The approach seems to have tackled the basic problem well. The students learn to pay better attention to their coding practices and they develop themselves good basic programming habits already on their first courses. After the first shock of having more requirements for the practical programming assignments, students have regarded the automatic style analysis as a useful aid in the programming courses. The quality of coursework has improved, and including a systematic assessment of programming style also improved the partitioning of contents and design of the courses involved. With more careful monitoring and evaluation of the tool usage, we hope to gain even better results in improving the understanding and adoption of good coding principles as part of students' personal programming practices.

Keywords: C++ programming style, automated assessment, programming courses, self-assessment

Editor: Linda Knight

Introduction

Programming style, i.e., the way to use a programming language and to write program code, is one of the important but too often neglected issues in programming. With habits of writing bad code, it is possible to make a program impossible for other programmers to understand. There may also be language-dependent coding requirements that need special attention for avoiding errors in program functionality. This is especially true for the C++ language, which has several pitfalls for the programmer, e.g., implicit type conversions and copy methods. By following good and systematic programming guidelines, it is possible to avoid these kinds of problems. Common guidelines and coding conventions are essential in real-life software projects, because the projects are usually carried out in groups, where several programmers should be able to work together effectively and pay attention to the quality of the software.

For novice programmers, it is often difficult to understand the relevance of programming style guidelines while they try to concentrate on producing their first programs. As noticed by Schorsch (1995), students commonly perceive programming style as secondary, not integrated to program development. Hence, they often clean up a program's style last, or not at all. A small program can run correctly even if it is badly written. The problems of unsystematic or otherwise bad coding habits demonstrate themselves in bigger programs, or in a situation where a program should later be modified by another person or ported to another environment. For this reason, it is essential that students always get feedback on these issues in their work. Students should not be let to use bad or inconsistent coding habits freely, because these may be very difficult to change later.

Programming courses generally contain lots of practical exercises: the issues to be learned do not become concrete for the student until tried in a program. The first programming courses aim at giving students the basic programming skills on which they can later build more advanced skills and knowledge. In these courses it is especially important to pay attention to programming style of, and to try to guide students to learn good programming habits. For achieving this goal, students should always be offered as good and thorough feedback on their work as possible. Unfortunately, assessing and giving feedback on these issues is time-consuming and difficult, even for experienced programmers and teachers. Large programming class sizes make the problem even worse. Often the only way to keep the level of required practical programming work and the amount of feedback offered for students on a reasonable level, is to develop software solutions to assist teachers in their tutoring and assessing tasks.

This article is organized as follows. The "Background" section reviews existing approaches to programming style analysis and assessment tools for it. The third section briefly describes the principles for developing C++ coding guidelines and the assessment tool for them. The fourth section presents the context and practices for using the assessment tool in programming courses. The fifth section presents some feedback and experiences from taking the tool in use in our courses. In the evaluation section, we discuss and evaluate our approach according to a framework defined for evaluating educational technology. Finally, we conclude the paper with some prospects for future work.

Background

Good programming style has been debated many times in developing programming languages. Heated discussions have taken place about whether, and in what matter, style can be quantified. Research on measurable programming style definitions was very active in the 1980's, when some well-known basic models and assessment applications were created.

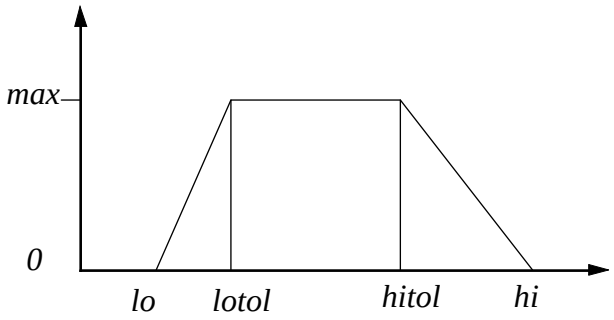

Figure 1: Marking scheme by Rees (1982).

One of the early works was published by Rees (1982), who developed a STYLE system to assess programming style of Pascal programs. He suggested that by making style issues visible and measurable for students they learn to pay attention to them. Rees proposed ten easily calculable measurements, e.g. percentage of comment lines, number of gotos, and average length of identifiers, and created a marking scheme that relied on five parameters, as illustrated in Figure 1. The scheme defines a range for desired measurement values with maximum mark, limits for giving zero mark and linear interpolation between these marks. For example, a recommended amount of comment lines in a small program may be set to 20%-40% of the program lines in total. The values named as *lotol* and *hitol* in the figure would represent the lower and upper bounds of this range, and maximum mark is given when the measured value falls in this range. A linearly interpolated value between maximum mark and zero is given for measurement values between *lotol* and *lo* (e.g. 10% in this case) or *hitol* and *hi* (e.g. 70%). If the measured value is lower than 10% or higher than 70%, no points are given.

Based on Rees' work, Berry and Meekings (1985) developed a style analyzer to check C programming style and tested it with 80000 lines of C program code. Their work has been the basis for many C program style assessment tools, e.g., Ceilidh (Benford, Burke, & Foxley, 1992) and modified for Ada in ASSYST (Jackson & Usher, 1997). In these systems, the parameter values for the measurements are usually configurable by the course teachers. In Ceilidh, the model is further refined to also include text comments for students, i.e., giving certain feedback if the measured value is below *lotol* or over *hitol.*

Later, Redish and Smyth (1986) developed a Fortran style analyzer to help with large classes. They approached the problem by classifying measurements under stylistic qualities like economy, modularity, simplicity, structure, documentation and layout. Their analyzer program contained altogether 33 measurements, including block sizes, number of operators, number of statements and complexity measures. Other programs for educational purposes have also been implemented, e.g., CAP (Schorsch, 1995) for self-assessment of syntax, logic, and style in Pascal programs, and Michaelson's (1996) analysis tool for functional programs.

Oman and Cook (1990) gathered information from different style guidelines and analyzers. They proposed a style taxonomy for categorizing stylistic factors into General programming practices, Typographic style, Control structure style, and Information structure style. They suggest that instead of a set of separate style rules, the students should be taught principles of programming style, e.g. according to their taxonomy. Their work has often been referenced in later educational approaches.

In software engineering research, Dromey (1995) published his model on connecting software quality attributes to program properties. PASS-C (SQI, 1993) can analyze 99 measurements from

C programs and gives feedback connecting the measured defects with software quality attributes. Similar analyzers are also implemented for Ada and Java languages. It is possible to see in his work the connections between overall software quality and different style rules. Although the approach is quite different, there can also be seen some similarities to the categories by Oman and Cook.

However, it is difficult to find assessment approaches for the C++ language from the literature. Some industrial guidelines for C++ programming have been published, such as by Taligent Inc. (1995), or Ellemtel laboratories (Henricson & Nyquist, 1992). These are usually high-level recommendations for object-oriented program design and coding in C++. Some experiments on automatic style measurements have also been implemented (Ayerbe & Vazquez, 1998). We could not, however, find any available assessment tools for C++ for educational purposes in the literature. We had tried to use the Ceilidh system with its C style analyzer, but considered it impractical for C++ programs. Old measurements that were based on statistical calculations from the program code did not cover many of the features that are important in object-oriented C++ programming. In addition, many of the previously important structural checking features were now obsolete, since these are automatically performed by the new compilers. The decision was made to design and implement a new style analyzer for C++ programming courses, taking into account the previously published research in the field.

Developing C++ Programming Style Assessment

When developing assessment practices for use across several courses, one has to pay very careful attention to them. The selected approach was to carefully collect information from literature and experienced programming course teachers, to combine all the knowledge that was used in everyday teaching situations. At the same time, we tried to pay attention to the flexibility of the solution.

Since most of the basic programming courses at TUT used the C++ programming language, a common C++ programming style guide for both students and faculty had already been created. It was mainly based on previous work by Cargill (1992) and Meyers (1992), industrial guidelines (Henricson & Nyquist, 1992), and the C++ language standard. A systematic common guide has been found to be a good approach as such, as reported by Poole and Meyer (1996). For our university, however, it did not have a noticeable effect on the students' coursework. This was mainly due to the large classes; tutors did not have enough time to check the style issues, although these were clearly specified. The style guide and its references, however, provided a good basis for developing an automatic assessment tool to really incorporate these issues into the courses.

The tool development project started by gathering programming course teachers of the institute to discuss the coding practices in general and the programming style issues the teachers had found problematic for the students in their courses. The requirements were prioritized according to the perceived relevance and implementability to form the basis for developing an assessment tool. From these, literature, and the previously collected guidelines, different programming style components were collected and placed on different abstraction levels. Following the example of Dromey (1995), the measurements on the program code level were designed and developed to correspond to software quality requirements. High-level software quality concepts were divided into subconcepts and factors until measurable program features on a concrete level were reached, when possible. Figure 2 illustrates the higher levels of the concept structure developed in the work.

The rules were gathered to give guidelines for developing reliable, maintainable, clearly structured and clearly written object-oriented programs with special emphasis on correct use of the risky features of the C++ programming language. To be able to measure and recognize these fea-

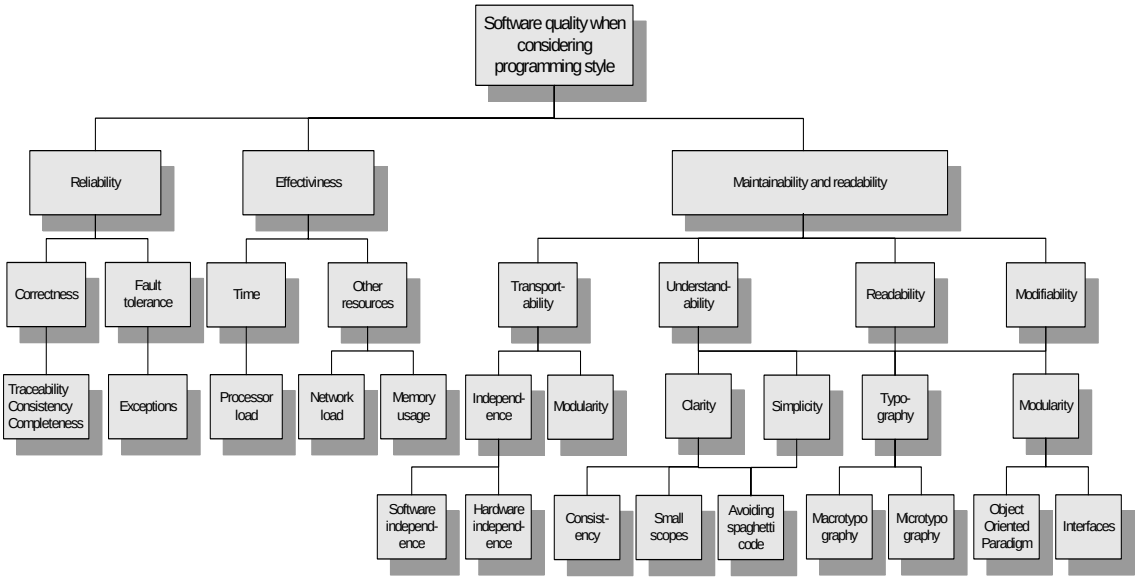

Figure 2: Style-quality structure for programming style (Uimonen, 2000).

tures, it was essential to have more knowledge of the program structure than obtained just by collecting statistics from the code. Thus, the analyzer is based on a C++ compiler front end that can provide information on the program structure and variable scopes. The main goal has been to assess the practical coding and design conventions of the programs. While some prioritizing was necessary, some advanced issues, like code complexity, have not been included in the measurements or the guidelines at the moment. Furthermore, the features that can already be analyzed by C++ compilers when given full warning options, including unused variables or parameters, and usage of only ANSI C++ compatible features, were decided not to be implemented.

The measurement features for the coding practices can be divided into the following main categories:

- Modularity, including various C++ class implementation issues, e.g. declaring explicit constructors and assignment operators, explicit calls for inherited assignment operators, and virtual destructors in inherited classes. Furthermore, general encapsulation issues, like usage of friend classes, public member variables, and pointers to private data belong to this category.
- Typography, including various measurements about commenting practices, naming conventions, and code layout issues.
- Clarity and simplicity, such as average length of functions, blocks with braces, and usage of short-circuit statements.
- Independence, such as avoiding numerical literals and correct return values.
- Effectiveness, such as small variable scopes.
- Reliability, such as float equality comparisons, default blocks in switch statements, and setting pointers to zero after memory block deletion.

One should 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. However, this does not reduce the value of teaching these issues at the university. On the contrary, for this reason it is all the more important for the students to learn to pay attention to their coding practices and to learn to follow and sometimes also criticize the given

guidelines. Furthermore, the basic ideas and core conventions are usually similar in all style guides. Thus, university courses should pay attention to these basic rules, to give a good understanding of their benefits and usage for all students.

Customizing Assessment for Different Situations

The tool covers altogether 64 basic measurements. However, it is clear that different courses want to emphasize different style practices. A subset of rules may be useless in some connection but essential in another, for example, inheritance rules in connection to a short introductory course on C++ programming as opposed to an advanced course on object-oriented programming. Therefore, the assessment should be easily configurable for different situations.

The adjustability is based on the metric boundary model, originally developed by Rees (1982) and presented in Figure 1. The teacher can set the measurement value limits for percentage or absolute values to define the requirements for each measurement. Different verbal feedback messages can also be set according to the measured values. The teacher can decide the contribution of each metric, since each feature has a weight that can be either added or subtracted from the total score. All these settings can be defined with a configuration file that can be provided to the tool as a parameter. If no special configuration is given, the tool uses a default configuration. At the moment, the default configuration follows the weighting principles developed for the introductory object-oriented programming course.

In practice, this means that the teachers have an automatic assessment tool for C++ programming style, for which they can themselves decide the following:

- which program style features are assessed,
- what kind of acceptance levels are set to each selected feature,
- what kind of verbal feedback messages are connected to each feature and its measured values, and
- what kind of weighting factors should be given to each feature in the total score.

With these, the teachers can tailor the assessment, for example, to the most relevant issues of the course, to the type and size of the programming task, or to the basic requirement levels of the course. Often teachers define the required levels by implementing a model solution and assessing it. More experienced teachers can just rely on their previous experience with the tool. A common approach has been to define large negative weight factors to the most important issues and noticeably smaller weights for the less important issues. When defining the minimum pass score appropriately, this strategy ensures that all crucial issues are followed, but some additional feedback is also given. When teachers release the configuration files for students to read and use, students also get an easy way to learn and evaluate the programming style issues that are the most relevant in their studies.

The assessment rationality of the tool is clear, so there are usually no “false negatives” in the results, if the teacher requires that all style rules are strictly followed. If some issue does not need to be followed, it can be turned off from the configuration file or given a very small weight. There is a couple of inconsistencies in the C++ language implementation that cause the tool to detect these as errors from the general rule. For example, all other member variable types except arrays can be initialized in the initialization list of the class constructor, and the general rule is that all member variables should be initialized there. This exception is not built in the tool yet, although we have planned to work on it. At the moment, the situation is handled by adjusting the assessment configuration to accept that, say, 5% of member variables are not initialized, if it is known that array member variables are needed in the program.

Example of Tool Usage

The tool, called Style++, has been implemented for the UNIX environment, which is the required environment for the programming coursework in the first courses. The output format of the program is designed to resemble compiler output, the basic tool for all programmers. The output format and content can be controlled by giving different options for the tool. For example, the output can contain either comments and scores of all assessment features or only from those that have defects. The output can contain only the given scores compared to the maximum, or also the information of the required measurement values. The exact requirements can also be checked from the released configuration file with teacher's commentation. If a student or a teacher is not interested in scores, it is possible to invoke the tool to just give all verbal warnings about the style defects in the program. There is also an X-window interface to the program, although most students use it from command line, just like the C++ compiler.

The limited space of the article does not make it possible to show an analysis of a normal-sized object-oriented programming assignment. However, we will try to give the reader some idea of the functionality of the tool with two versions of a (very) small program, presented in Figures 3 and 5. These programs ask from a user the value of the absolute zero temperature in Celsius degrees and print whether the answer was correct with 0.01 precision.

```
#include <iostream>
int a;
int main(){
   std::cout<<"Give absolute zero temperature in Celsius degrees: ";
   std::cin >> a;
   a==-273.15?std::cout<<"Correct!":std::cout<<"Incorrect.";
}
```

Figure 3: Example program (example1.cc) with many style defects.

```
Scanning file 'example1.cc'

Warnings:
File: example1.cc
  ABLP: No blank lines found in file.
  ACLP: No comments found in file.
  CABP: No beginning comment.
  AOGV, line 2: Variable 'a' is global.
  IGLP, line 2: Identifier 'a' is too short.
  IVAP, line 2: Variable 'a' is not initialized.
  RVOM, line 3: main() function return value improper.
  AOCE, line 6: Conditional ?-operator found.
  AONC, line 6: Numeric constant bigger than 10.
  EOWF, line 6: Floating point equality/inequality operation found.

Score table                          P1    P2   P3    P4 Value  Score  MaxP
-------------------------------------------------------------------------
Amount of blank lines                 0    15    60   100     0    0    10
Amount of comment lines               0    25   120   180     0    0   100
Amount of conditional ?-expressions   0     1   100  1000     1  -50   -50
Amount of global variables            0     1   100   100     1  -100  -100
Amount of numeric constants           0     1   100   100     2  -100  -100
Comment at beginning of file%         0     0    90   100     0  -100  -100
Float equality comparisons            0     1   100   100     1  -100  -100
Initialized variables                 0    95   100   100     0    0   100
Return value of main()                0     0     0     1     0  -50   -50
-------------------------------------------------------------------------
Total score: 80 / 790 ( 10% )
```

Figure 4: Example output from Style++ analyzer for example1.cc.

Figure 4 shows an example feedback given by the style analyzer on the program code in Figure 3. In this example, the tool is set to print only the warnings and the scores of the style requirements that were not followed according to the requirements. Thus, it does not list the accepted features, but they are included in the total score. As can be seen, in this example there are so many high-weighted subtractive scores that the total score obtained is very low.

Figure 5 presents a corrected program; this version gets full score from the Style++ analyzer.

```
// A small program to ask user the value of absolute zero
// temperature and to print whether the answer is correct.

#include <iostream>
#include <cmath>

const float ABS_ZERO = -273.15;  // absolute 0 in celsius degrees
const float PRECISION = 0.01;    // precision for correct answer

int main(){
   float read_value = 0;

   // ask and save user's guess for absolute zero
   std::cout << "Give absolute zero temperature in Celsius degrees: ";
   std::cin >> read_value;

   // check and print whether answer was correct with required precision
   if( std::abs(ABS_ZERO - read_value) < PRECISION ){
      std::cout << "Correct!";
   } else {
      std::cout << "Incorrect.";
   }
   return 0;
}
```

Figure 5: A new version of the program with corrections of the recognized style defects.

Programming Style Assessment in the Student's Study Process

Practical programming exercises are an essential part of any basic programming course and often provide most of the learning experiences in the course. Thus, students should have the best possible support for working on them. As opposed to the old problems in program style teaching, with an automatic evaluation tool it is possible to offer students continually available, instant, consistent feedback on their programming practices and style issues. Different courses use style assessment in slightly different roles. However, the main idea in all of them is to offer the tool and the relevant assessment configurations freely for students to use anytime they want.

Students work on their coursework usually independently, at their own pace. When a student has completed the assignment and believes it to conform to the given requirements, he/she can invoke the style analyzer tool to get instant feedback on the programming style features of the program. The verbal and numerical feedback from the tool helps the student to improve the assignment. In many courses, there are also other automatic evaluation tools, so that the student can get automatic evaluation of, e.g., the dynamic memory use of the program. In some courses, Style++ is connected to Ceilidh system (Benford, Burke, & Foxley, 1992) that can also check the functionality of the program.

Finally, when a student decides that the coursework is finished and in every way as good as he/she can make it, he/she can submit it to the course personnel for further evaluation. The nor-

mal procedure for the course personnel is to run the automatic evaluation to check that the basic requirements really were met and then concentrate on giving personal feedback on the more complex and advanced issues of the program, which cannot be evaluated automatically. For example, the sensibility of naming and comments, or the goodness of class design for a specific task require evaluation by a person with programming experience. The whole coursework process is illustrated in Figure 6.

The goal of this practice is to teach students to follow good coding practices when developing their solutions and to use automatic tools if they are unsure about their own abilities to evaluate whether their work conforms to the given requirements. We hope that by requiring good coding practices already from the very first courses, the students realize for themselves the benefits of using these systematic guidelines in their own work.

Students' work is supported by written explanations about the tool's evaluation principles, linked to the relevant style rules and their reasonings, all available on WWW. The courses have normal lectures, exercise sessions and assistant appointment hours for face-to-face discussions. In our programming courses, the assistants also actively respond to questions in the course discussion group and to students' emails. In the very first courses, the style assessment is often a part of the official coursework acceptance process, as presented in Figure 3. In other courses, the tool usage is not that strict and Style++ acts mainly as a supporting tool in developing students' consideration for program style issues. In these courses, students are required to evaluate their program against the given requirements. However, they are allowed to depart from the given guidelines, if they provide a good justification for their differing solutions. This develops the students' abilities to analyze for themselves the best programming practices in possibly exceptional circumstances, with the knowledge of general solutions as a basis for their decisions.

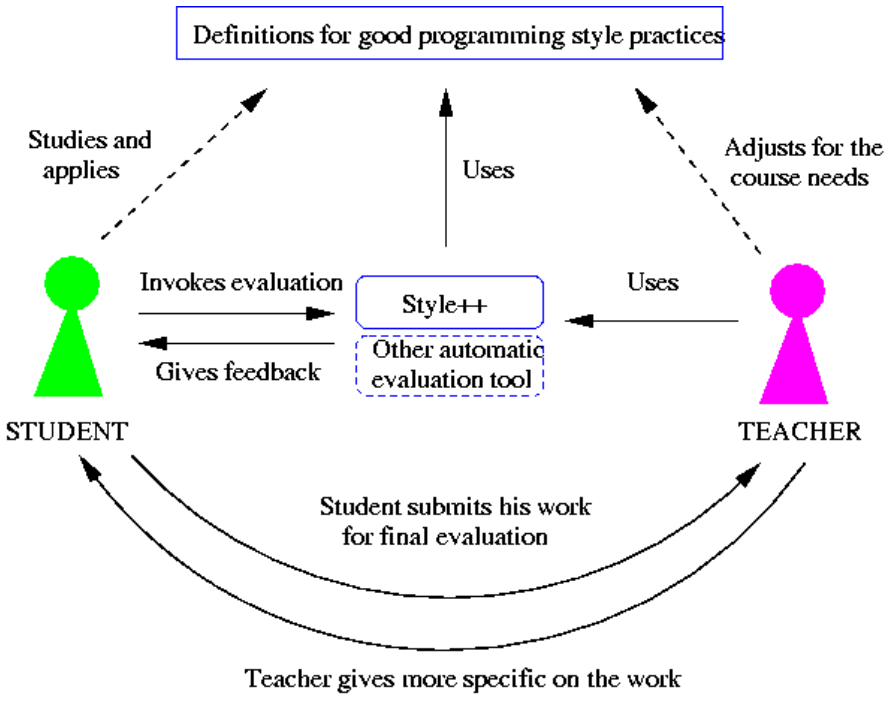

Figure 6: Coursework process with automatic style analyzer Style++.

Using the Tool

Style++ has been used in several programming courses since the year 2000 and has been used by over 1500 students. Experiences and feedback on tool usage have been collected by different kinds of questionnaires from students and in personal discussions with them over the years. The teaching personnel have also been interviewed about their experiences in using the tool and in tutoring students who have used it. Another valuable source of feedback and user opinions have been the course specific UNIX newsgroups that contain discussions about course related issues and problems with the coursework. The discussion messages always provide good insight to the most difficult and the most interesting issues in the course.

Integrating style assessment to the course contents raised some resistance among the students. With an automatic tool, teachers could systematically follow that the basic requirements were met in all students' coursework. It could be seen that previously, without systematic checking for programming style guidelines, many students had not paid much attention to programming style issues. Now, these students saw it as extra work required from them and protested loudly. This phenomenon was apparent in the first courses where Style++ was used. These courses were aimed at second year students, who had had the possibility to pass previous programming courses without systematic program style assessment. The complaints were often based on the statement that teachers should not change the requirements of the course from the previous year if they are making it harder for the present students.

Most complaints and resistance about program style guidelines and assessment were presented during the first year. We collected an anonymous questionnaire survey from the data structures and algorithms course students after about 2/3 of the course was passed. Although the course discussion group had mostly negative comments about the tool from students, the results of the survey showed the opposite, even though there were still some technical problems with the new tool. The results are presented in Figure 7. As can be seen, most of the students considered the evaluation tool as a good solution and means to support learning new issues in programming. However, it can be seen that many of them thought that the required measurement levels were too fastidious. This was taken into account when developing new, more forgiving assessment configurations for the course next year.

After the first courses, the students have presented negative opinions and discussion messages only rarely and no special problems have been noticed from the course feedback either. Usually the critics of the required coding conventions have been older students with a lot of coding experience even before beginning their studies at the university. A similar phenomenon was also noticed by Schorsch (1995); experienced programmers do not want to change their habits for programming courses. But all in all, general feedback on Style++ and its incorporation into courses has been positive, both by students and course personnel. It can be said that the culture of learning programming in our institute has evolved to contain programming style issues as a natural part of the learning contents. Also the authors of Ceilidh (Benford, Burke, & Foxley, 1992) recognized a change in the learning culture with their assessment system; students became better aware of the quality issues when these were systematically controlled.

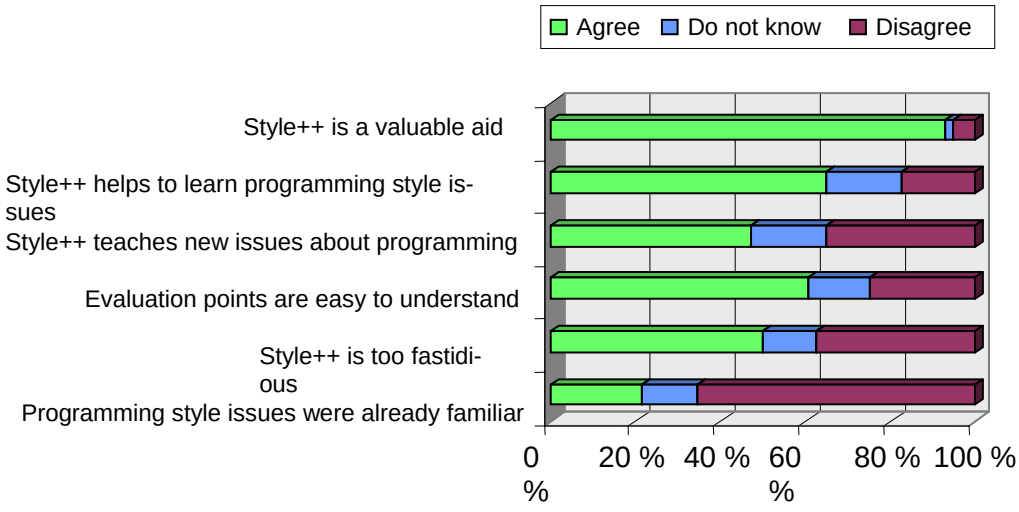

Figure 7: Student feedback on Style++ (n=61).

Students' initial resistance was often due to the fact that they did not understand the relevance of the given programming style rules. Some students cheated on purpose and some accidentally used practices that misled the automatic assessment. However, since all coursework was also evaluated by tutors, these students later got feedback on these matters, too. These cases showed that the machine-run evaluation cannot be perfect and that it is necessary to have also human tutors. Students and teachers had many discussions about the meaning and relevance of certain style requirements. These discussions showed that the reasoning for different requirements has to be made very carefully in order to motivate the students to learn new practices that they perhaps do not see as relevant. It was also noticed that all students did not utilize the guiding documentation and FAQ lists as much as was expected. They often did not read the documents carefully or at all, and there were also problems with understanding the real meaning of some written rules. Therefore, it was found very important that the teacher brought these up in other contexts, too.

Evaluating the Presented Practice

Our approach to the problem of learning programming style issues includes guidelines and a tool-supported assessment process integrated to the practical essence of the programming courses. Although the term "educational technology" often refers to large educational systems, we suggest that a simple technical tool can also be considered as an important educational technology solution. However, this requires careful design and evaluation of the approach, both from technical and pedagogical points of view. The technology needs to be evaluated in terms of its effective use in education. To emphasize this viewpoint, we discuss our approach following a framework for quality in educational technology programs, developed by Expert Panel of Educational Technology (Confrey, Sabelli, & Sheingold, 2002). The main viewpoints of the framework are Quality of the solution, Educational significance, Evidence of effectiveness, and Usefulness to others. We will not go into specific details or grading levels, but try to describe the whole tool-supported process in the terms of and by the requirements set in the framework.

Quality of the Solution

The solution should address important educational issues: Programming style, especially common coding conventions are very important issues in software projects in practice. Taking these issues to university education will better prepare students for the working life. Students who learn these issues early get a good understanding and routine of important programming practices already at the beginning of their programming career. Style++ implementation relies both on research in programming style principles and on practical experiences from programming educators. An automatic evaluation tool itself was a necessity to assure that the given guidelines are followed in the practical programming work of all students. Although the assessment is carried out by a set of measurable rules, the main purpose is to teach students the underlying principles and the connections of coding practices to software quality.

The usage of the tool in our courses was described in Figure 6, its main role being to support students' independent work on programming projects, which form the essence on many programming courses. The ideas and motivations for good programming practices are introduced in contact education situations and carefully documented textual guidelines. Actual learning happens when students design and implement their programs, trying to apply good coding practices in the process. They can use the tool for getting feedback on how well they have succeeded and which issues they should pay more attention to.

The tool is offered freely for use on the campus computer network, to which all students and faculty have access either via modems, ISDN, ADSL, or cable connections. In addition, all student dormitories and the workstations in the campus area are connected to the network. The tool does not need continual maintenance and the user support is mainly taken care of by course personnel together with the FAQ lists and the tool usage documentation published on WWW pages.

Educational Significance

The Expert Panel considered learning, equity and organizational change as essential areas that need to be addressed in the evaluation of educational significance. First, the solution should develop complex learning and thinking skills. Second, the program should contribute to educational excellence for all equally. Furthermore, the solution should promote coherent organizational change.

Learning

With the presented process, students learn practical programming guidelines already from their first courses. They learn to apply these guidelines in their programming practice, and they adopt programming habits that help them concentrate on the more demanding issues in software design and implementation in advanced courses. But, it is possible that some of the students do not get in-depth understanding of the value of these practices at the beginning and just follow the instructions without thoroughly understanding the reasons. However, examples show that also these students can learn to understand the relevance of these issues after having more programming experience and seeing the benefits of the practices they have used. Redish and Smyth (1986) already suggested that since students are humans, they will construct a meaning with their human intuition for the style rules they are following.

Computer science students are usually well prepared to deal with technology. It is necessary, since many phases of software projects are supported by different tools and applications. Style++ also prepares students for this, being an example of a tool that needs to be taken into the development process. The programmer has to learn an ability to study his own program, recognize the part that is not implemented correctly or according to given specifications, and develop a solution

to fix the problem. Furthermore, students must learn to be critical and to consider the possibility that general specifications or tools may be imperfect in some special situations. They can also practice these skills with Style++ in more advanced programming courses.

Since taking Style++ in use has taken place gradually and caused changes in the teaching practices as well as inserted programming style specific contents to courses, it cannot be systematically studied how much effect it has directly had on students' learning. We approached the issue this year by organizing a student survey on a software project course, which is usually taken after all basic programming courses have been completed. We asked the students their opinions, based on their personal experiences, about various tools that are offered at our institute for programming coursework. There were also specific questions relating to Style++, asking whether students perceived it useful (a) when working with the coursework, or (b) for their learning.

Figure 8: Student feedback on the usefulness of Style++ (n=88).

The results of these questions are presented in Figure 8. The answer choices ranged from 1 (completely useless) to 7 (exceptionally useful). As can be seen, there is a strong correlation between the perceived usefulness and the amount of completed basic programming courses with Style++. Since all our basic programming courses nowadays use Style++, a low number of these courses implies that a student has studied programming less than the others, or in a few cases, many years ago. The results support our goals in that students learn to appreciate programming style issues. The verbal comments also supported this perception; students with less experience criticized style rules etc., while students with more experience hoped that Style++ would also be available for other programming languages. The results also show that students do not only perceive Style++ as a useful tool in preparing their work, but have experienced that it supports their learning.

Equity

With common guidelines and requirements for all students, Style++ treats everyone equally. It has removed the old problems in having several assistants in a course for giving feedback and assessing the coursework. Even with precise instructions given to the tutors, they were individuals, so there was no way to assure that all students were given feedback equally and assessed consistently. Now the requirements are undeniably the same for all and are evaluated as quickly every time, no matter at what time of day students send their program for evaluation. However, teachers can adjust the requirements according to the level of the exercise and the students, if necessary. This can be also used to take into account other different kinds of situations, e.g., international students that need feedback messages in another language.

When considering different learning styles, it can be seen that this solution favours students that are highly motivated, self-controlling, and have good learning skills, while previously all students were more on the same level. All students have possibilities to ask for personal support in newsgroups, by email, during appointment hours, or in contact education situations. However, the students that have an ability to adopt and search information from the given instruction documents on the WWW are better served since they can get information quickly at any time of the day. This hopefully also motivates students to learn more self-directed learning and information searching skills.

Organizational Change

Style++ has had an effect on courses also in other aspects than bringing more content to learning. Systematic style evaluation showed some weaknesses in the way the teachers had dealt with and presented programming style issues in the courses. The style guidelines contained features that could not be understood by the students, since they had not studied the underlying issues in the prerequisite courses. The different programming courses had commonly agreed interfaces, but these had got outdated with continuous course development. The common program style evaluation tool brought up these issues and the teachers noticed the need to co-operate more in course development work.

Good experiences with Style++ in the students' learning results have made more teachers willing to incorporate the tool into their courses. Many courses have now an automatic preassessment phase, where the student work is evaluated against the given minimum requirements. Because of the automatically conducted assessment before coursework submission, the tutors can concentrate their work on giving feedback on the more advanced issues. This has had a clear effect on the contents of the tutors' work and on their ability to teach students more in-depth issues of the essence of the course. Nowadays programming style issues are monitored systematically on all basic programming courses.

When monitoring Style++ users and their questions about using the tool, we noticed a need for a central user support for the tool. Course personnel usually answer most of the questions, but it means that several tutors in different courses need to concentrate on the usage of the evaluation tool and the typical problems that students have with it. Their workload could be reduced by designating one person for central user support to maintain FAQ pages, to answer general questions about the tool and to forward course related questions to the tutors. There are also many other automatic tools in the software engineering and computer science courses of the institute. Thus, we developed a practice where all educational tools have supporting WWW-pages and a special support person. The same person also takes care of tool maintenance when necessary. This improves the level of support that is offered for both students and faculty in using programming related tools in our institute.

Evidence of Effectiveness

Obvious benefits have been received in that the students now follow many important programming style guidelines. For example, students now always initialize their variables and write comments in their program code. Especially for the C++ language with many dangerous default functionalities, we now have a tool to emphasize to the students that they always need to take into account, e.g., the need for self-defined copy constructor or assignment operator for their classes. Class interface design can be guided with rules that prohibit public member variables or friend definitions for classes. When comparing accepted programming assignments from the year before and after taking Style++ into use, it can be seen that the students implement more reliable and understandable programs than before. Table 1 presents example figures of programming style

related errors that were found when reassessing old, accepted, assignments before using Style++. The same style rules were applied at that time, but many errors missed the eye of the teaching assistants or were knowingly let to be. TAs had usually time to comment and require resubmission of only the biggest style errors, and as a result, 86% of the then accepted assignments still contained style errors. All of these errors can now be completely removed or restricted with Style++, depending on the assessment configuration designed by the teacher of the course.

Table 1: Examples of style errors in accepted programming assignments before Style++, n=192

Style defect	Frequency	Present practice (typical Style++ configuration)
Uninitialized variables (incl. member variables)	83% (159 assignments)	at least 95% of all program variables need to be initialized, this is controlled by a high weight factor
Hardcoded numeric constants in program	11%	prohibited by a high weight factor
Deleted pointers not always set to zero	77%	guided by a moderate weight factor
Missing default blocks from switch statement	18%	guided by a moderate weight factor
Global variables or public member variables	3%	prohibited by high weight factors
Commenting practices	50%	4 separate measurements guiding for good commenting practices, e.g., requiring beginning comments and guiding for suitable comment line proportion in program

According to the teaching assistants and teachers, the quality of students' coursework submissions has got noticeably better after Style++ was taken into use. The program implementations are now more clearly structured, contain less "bad" C++ language usage, and above all are well commented and laid out before they are submitted to the teaching assistants. This means that they can more easily read and understand the students' solutions and concentrate on giving feedback on those program features that should be the essence of the course, like module and interface design, data structure implementations, and general program architecture.

It has not been measured, however, how much of the obtained coursework quality level is usually achieved by the student on his own, and how much is due to the automatic tool that pinpoints the problematic issues for the user. The same problem was also recognized by Schorsch (1995). From the submission tool logfiles it can be seen that some students do not evaluate their work themselves very much but give it several times to the tool and correct only the issues that are pinpointed by it. However, it is likely that they also try to follow most of the guidelines already when creating the program since they know that at the end they have to do it anyway. This issue will be further studied in the near future by collecting information from tool usage and interviewing students to understand their reasoning for different ways of using the tool.

Usefulness and Availability

Style++ is implemented to be flexible in use and portable to other environments. Although the tool is implemented for the C++ language and contains fixed style measurement points, its architecture is designed so that it is possible to add new evaluation features in it or to re-use the architecture in a style analyzer for some other programming language. So far, there already exists one new version of the tool in C++ that has been extended to cover Symbian specific measurement points (Halonen, 2003). With Symbian development, the special coding and naming conventions (Symbian DevNet, 2003) have a great impact on the quality of the software, since they also affect the functionality of the program. Even software companies have been interested in taking these assessment tools in use for software quality checking purposes.

The largest time investment has been designing and implementing the basic solution: maintenance costs are low. Style++ is based on the C++ Front End provided by the Edison Design Group (1992) that is available for non-commercial use for other educational bodies with a special licence. Style++ is also freely available for non-commercial use and does not require specific maintenance. Unfortunately, the software documents for the Style++ are available only in Finnish at the moment.

The overall solution of developing requirement definitions to a discipline-specific quality problem and taking them actively into the education should work also in other disciplines. The basic requirement is that measurable quality factors can be defined for the issue and discipline in question. With automatically measurable factors, it is possible to implement a tool that can be offered for students and teachers. It has to be remembered, however, that just efficient assessing does not cause learning. The students need to be offered enough information about the learning objectives so that they get motivated to learn and apply the issues and to use the provided tool for evaluating their success.

Conclusion

In this article, we have approached the problems of teaching programming style and coding conventions in programming courses. We have built a set of common guidelines and worked them into automatically measurable program style factors. An automatic style evaluation tool Style++ was implemented and incorporated into courses to support students in their efforts to develop good basic programming routines, which are necessary for more advanced programming tasks. As an educational technology solution, this tool is a very simple approach. However, the importance of programming style and the possibility to firmly incorporate it into courses with this tool have added lots of relevant practical contents to the programming courses and had a positive impact to the quality of coursework. It has also made it possible for the teaching personnel to concentrate better on the more advanced evaluation and tutoring issues in the courses.

When evaluating our solution, it has to be kept in mind that measurement tools do not guarantee in-depth learning. The teachers must incorporate style issues and the motivation for them into courses so that the students see their relevance and gain motivation to learn them. Students should always have available guiding documentation as well as personal assistance and guidance. Students' learning behavior could be more carefully observed by recording the usage of the evaluation tool. This is the next step when we study development targets in the present teaching methods and the educational usage and impact of automatic assessment tools.

The technical solution seems to be adequately flexible for different education purposes. There are, however, some usability and technical issues that could be further developed. Software companies have showed interest for the tool because of its undeniably positive effect on software quality. It has been shown that the tool can be extended to different software platforms with spe-

cific reliability and code convention requirements. All technical advancements in the tool development and new results from the program style research will also be included in our courses, to support learning with best possible knowledge and practices.

Acknowledgements

Authors wish to thank Edison Design Group for offering their C++ compiler front end (EDG, 1992) for research and educational use to form the core of the Style++ analyzer. They also thank Mikko Tiusanen and Reino Kurki-Suonio from Tampere University of Technology, and Janet Carter from the University of Kent at Canterbury for their careful reading and comments on the manuscript.

References

Ayerbe, A., & Vazquez, I. (1998). Software products quality improvement with a programming style guide and a measurement process. *Proceedings of the 22nd Annual International Computer Software and Applications Conference*, 172-178.

Benford, S., Burke, E., & Foxley, E. (1992). *Courseware to support the teaching of programming*. Retrieved April 7, 2004, from http://www.cs.nott.ac.uk/~ceilidh/papers.html

Berry, R.E., & Meekings, B.A.E. (1985). A style analysis of C programs. *Communications of the ACM, 28*, 80-88.

Cargill, T. (1992). *C++ programming style*. Addison Wesley.

Confrey, J., Sabelli, N., & Sheingold, K. (2002). A framework for quality in educational technology programs. *Educational Technology, 3* (42), 7-20.

Dromey, R.G. (1995). A model for software product quality. *IEEE Transactions on Software Engineering, 21*, 146-162.

Edison Design Group. (1992). *C++ front end*. Retrieved April 7, 2004, from http://www.edg.com/cpp.html

Halonen, H. (2003). *Automatic code convention checking in symbian environment.* M.Sc. Thesis, Institute of Software Systems, Tampere University of Technology. In Finnish.

Henricson, M., & Nyquist E. (1992). *Programming in C++, rules and recommendations*. Ellemtel Telecommunication Systems Laboratories.

Jackson, D., & Usher, M. (1997). Grading student programs using ASSYST. *Proceedings of the Twenty-Eighth SIGCSE Technical Symposium on Computer Science Education*, USA, 335-339.

Meyers, S. (1992). *Effective C++: 50 specific ways to improve your programs*. Addison Wesley.

Michaelson, G. (1996). Automatic analysis of functional program style. *Proceedings of Australian Software Engineering Conference*, Australia, 38-46.

Oman, P.W., & Cook, C.R. (1990). A taxonomy for programming style. *Proceedings of the 1990 ACM annual conference on Cooperation*, USA, 244-250.

Poole, B.J., & Meyer, T.S. (1996). Implementing a set of guidelines for CS majors in the production of program code. *ACM SIGCSE Bulletin, 28* (2), 43-48.

Redish, K.A., & Smyth, W.F. (1986). Program style analysis: A natural by-product of program compilation. *Communications of the ACM, 29*, 126-133.

Rees, M. J. (1982). Automatic assessment aids for Pascal programs. *SIGPLAN Notices, 17* (10), 33-42.

Schorsch, T. (1995). CAP: An automatic self-assessment tool to check Pascal programs for syntax, logic and style errors. *Proceedings of the 26th SIGCSE technical symposium on Computer science education*, USA, 168-172.

Software Quality Institute. (1993). PASS-C overview. Retrieved April 7, 2004, from http://www3.sqi.gu.edu.au/passc/

Symbian DevNet. (2003). *Symbian OS C++ coding standards*. Retrieved April 7, 2004, from http://www.symbian.com/developer/techlib/papers/coding_stds/2003-01_SyOSCodStn.pdf

Taligent Inc. (1995). *Taligent's guide to designing programs*. Retrieved April 7, 2004, from http://pcroot.cern.ch/TaligentDocs/TaligentOnline/DocumentRoot/1.0/Docs/books/WM/WM_1.html

Uimonen, T. (2000). *Automatic style analysis in software*. M. Sc. Thesis, Institute of Software Systems, Tampere University of Technology. In Finnish.

Biographies

M.Sc **Kirsti Ala-Mutka** acts as researcher and lecturer at Tampere University of Technology, Institute of Software Systems. Her main research interests are related to developing tools and processes for utilizing automated assessment to support students' learning in programming courses. Another interest lies in promoting co-operation and courseware interoperability between computing educators.

Toni Uimonen completed his M.Sc Degree in Computer Science in 2001 at Tampere University of Technology, with a specific research focus on C++ style assessment. He is presently working as project manager in Solita.

Dr.Tech. **Hannu-Matti Järvinen** is professor of embedded software at Tampere University of Technology. He is also the Dean of the Department of Information Technology at TUT. His interests contain teaching, operating systems, real-time and embedded systems, and fault tolerance.