A Process Model for Quality guided Programming

An Approach to Making Quantitative Evaluation of Software Systems Useful for Practitioners

STEFAN BIFFL THOMAS GRECHENIG

Department of Software Engineering,
Technical University of Vienna,
Resselgasse 3/2/188, Vienna,
A-1040 Austria, Europe

EMail: Biffl@eimoni.tuwien.ac.at
Tel.: ++43-1-58801-4082
Fax: ++43-1-504 15 80

Abstract

Quantitative evaluation of software systems has not yet been accepted by practitioners. Early expectations especially into code analysis have not been met so far. Among several reasons for the rare use in practice we suppose a lack of empirical data, a dominant focus in research on formal aspects, an unreasonable embedding in the development process. The following paper deals with more technical reasons: lack of flexibility and usability of code measuring tools.
We outline a process model for quality assurance during the coding phase providing human reviews as well as quantitative evaluation. The model is based on the idea of permanently adapting measuring tools to the goals of a project which will result in a metric and review guided coding cycle. The system presented generates software measuring tools providing the necessary flexibility for quick adaptions at hand. The generator is equipped with a clear separation of language and metric description making both reusable when a new tool design is being generated. Experiments with several commercial programming languages and most classical code metrics proved the claim of flexibility and usability.
We postulate that quantitative evaluation can work in practice if metrics, project constraints and management goals are matched within a local process of collecting empirical data.
Keywords: *quality assurance, software metrics, quantitative evaluation of software, tools for metric analysis*

1. Introduction

Software quality assurance (QA) has been a serious issue for industrial developers for at least ten years now. Nevertheless quality assurance groups have never been provided with essential rights to interfere in actual development. The typical QA group is a team of rather learned people whose specific knowledge is not used to an appropriate extent, though. Only within a few conscious development environments at least software document reviewing has been established. The acceptance of QA methods in practice is at the lowest leven concerning quantitative approaches: Software metrics
- could not provide the project information expected and
- therefore were scarcely applied.

Meanwhile research has been done to overcome the gap between the theory of software metrics and their application. [3], [25] suggested process models for implementing quantitative quality assurance. Some case studies on practical experiences have been reported (e.g. [20], [15], [32]). Nevertheless mainstream research still behaves as if it were necessary to market the use of software metrics as such ([35]). This is a clear sign that despite the work mentioned above, metrics and especially code metrics have still not yet proved to be successful in practice. Among the various reasons we presume
A focus on formal aspects: many collections of formulas have been suggested (e.g. [26], [1], [10] or [37]) that should map software characteristics like complexity, information contents, structure of data flow, structure of control flow. Obviously most of the metrics that have been suggested during the last 15 years are of a certain mathematical and statistical beauty. But they did not really bridge the gap to

the needs of software engineers at work. Practitioners want project oriented information. Mostly this is much more than aggregating measures on single code lines.
A lack of empirical data: though many experiences have been reported, no general theory has been derived from them. Probably this is impossible within the variety of today's development conditions. Results are reliable only within a certain environment and cannot be transferred easily. At least up to now no parametrized theory based on empirical results has been derived, that would allow for a reasonable customization and application in another development environment.
The embedding in the development process: In practice code metrics as means of quality assurance are applicated usually before a piece of code is on the real job but after fulfilling functional requirements and testing. This is why many developers feel about code metrics as if they were some unnecessary appendix. Quantitative requirements to code have to be embedded in the development process together with other quality assuring activities like e.g. a reviewing procedure. This process serves the purpose of quality guided development which can establish a sort of local standard ([12]).
Though it is often expected, it is impossible to gather information which is above the level of static semantics, e.g. on the quality of mnemonics or the readability of code comments, relying on code metrics only. Code metrics are no substitute for other quality assurance activities.

Metrics are most effectively defined, adapted or modified by each quality assurance group on their own by collecting *local* information and extracting *local* standards. One condition is a certain comfort in defining metrics and actual measuring. In fact this uncovers another important but underestimated technical reason for practitioners' low acceptance: **a lack of flexible tools.**
The following paper deals with a tool that supports the generation of code measuring tools in a flexible way. It serves the purpose of quick and simple definition, adaption, and modification of code measuring tools used during the development of a SW-project. The concept provides independence of metric and language definition. Usually measuring tools are designed for one specific language (one specific compiler) and a certain set of metrics. Separating the descriptions of language and metrics from each other makes

- the metric descriptions reusable and
- the number of definitions just growing linear with respect to the number of languages and metrics.

Part 2 of this paper deals with a quality assurance process designed for adapting code metrics to local conditions. A generator for measuring tools, that enables the quality assurance group to make quick tool changes and its internal technical solution is described in part 3. In chapter 4 the actual generation of a tool is presented and some experiences are reported. Part 5 outlines an example of using a measuring tool for metric guided development. Above all, the key issue to an actual use of quantitative techniques is their organizational embedding.

2. A process for creating, assuring and standardizing code quality

We mentioned above that quantitative code analysis will be more successful if

- it is applied together with other quality assurance activities,
- it is properly embedded in the process of quality guided development with respect to technical and organizational aspects,
- it focuses on gathering information and assuring standards which are *local* with respect to a certain domain of projects within a developer's business environment.

In the following we sketch a practical model for both quantitative and qualitative analysis which defines a process of quality guided programming. It will make clear that a flexible tool for code measuring is only one piece in a bigger mosaic. Though it is a necessary condition.

Adjusting metric definitions to project goals and standards is an iterative process collecting experience by measuring actual code and slowly minimizing the gap between the abstract project goals on one hand and the measuring results (metric values) on the other. It is a process of matching metric values and project constraints which should be guided and performed by the quality assurance group. Speaking more generally, this cycle process serves as a mediator between the project management and its programmers.

Nevertheless, it is unreasonable to derive information on code quality from code metrics only. We regard quantitative code analysis as one means of quality assurance. In the following the process of reviewing code by team members and experienced senior programmers is taken as an example for further quality assuring activities among others. Nevertheless we will focus only on metrics for details.

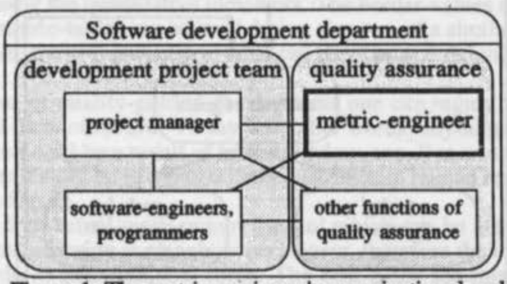

Software development department
development project team
quality assurance
project manager
metric-engineer
software-engineers, programmers
other functions of quality assurance

Figure 1: The metric-engineer 's organizational embedding in a software development department

We outlined that metrics will be at their best if applied locally with respect to determined types of projects and organizational and economic premises. To build up and maintain significant information requires a person, who is close enough to the software development team to be sensitive about their needs on one hand, but who is on the other hand not involved so much as to get overrun by the project's milestones and deadline pressures like it often happens to the programmers.
We call this position metric-engineering. The metric-engineer is a person within the quality assurance team (see fig. 1). He/she supports the project manager in planning quality issues and measures the actual software which programmers produce. He/she holds the same organizational position as e.g. the person who is responsible for the process and quality of code reviewing. Collecting information through metrics needs an experienced software-engineer with supplemental knowledge on formal methods of quality assurance and a solid background in literature of software metrics and models close to that.
Figure 2 shows a code quality assuring process in which the metric engineer's holds an important position. Three roles have to be considered: The project-manager (PM) serves as an expert for the goals of the whole project, a metric-engineer (ME) knows about metrics and models for rating the possible alternatives to achieve the project's goal and the software-engineers (SE), who implement the product following the technical model and a set of given metrics.

In the following paragraphs the steps in figure 2 are briefly described. A practical example is discussed in chapter 4.

Analysis of goals: The process starts with the PM's analysis of the project's goals (e.g. reusable code, high quality documentation, meeting deadlines, etc.), identifying the range of possibilities to achieve these goals and setting levels of necessary quality requirements for these alternatives.
Goals/questions/metrics: The PM and the ME meet to deduce questions, criteria and quantifiable categories from the goals, which can yield quantitative information about the achievement of the goal. From this catalogue of questions a set of metrics will arise, which has to be measured in order to answer the questions. On the basis of these metrics the possible alternatives to solve the problem get evaluated, cost and benefits are being estimated (GQM, see [3]).
Selection of metrics: The PM and ME decide, which alternative to realize as their first choice. For each metric identified in the previous step, they establish value-tables, which should be met when the task is finished: minimum, maximum, and a favorite value. The metrics get weighed, as to make the relative importance meeting their values obvious. At the end of this step the measuring tools are generated.
Metric guided programming: Software engineers and programmers are informed about their subtasks and the metric suit defined before. Usually the metric engineer provides them with the generated measuring tool. The software engineers start planning, designing, coding, and testing. They

should know the principal goals of low maintenance costs and fast reaction to bugs identified at the time of production.

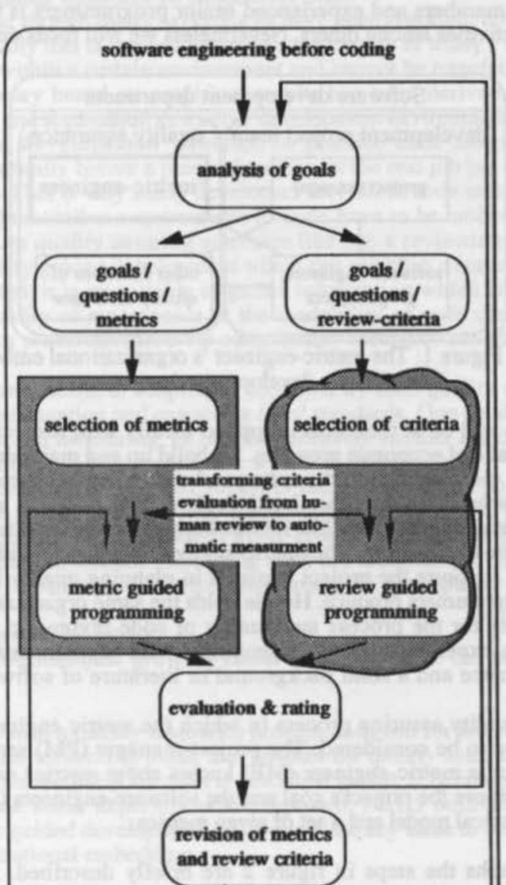

software engineering before coding
analysis of goals
goals / questions / metrics
goals / questions / review-criteria
selection of metrics
selection of criteria
transforming criteria evaluation from human review to automatic measurment
metric guided programming
review guided programming
evaluation & rating
revision of metrics and review criteria
software engineering after coding

Figure 2: A process for gathering, adapting and controlling standards for quality oriented coding

Evaluation and rating: While the SE works on the solution of his problem, the ME measures or helps to measure the newly developed pieces of code and provides feedback to the SE such us to enable him to adapt his style of working to the metric-tables if necessary. Partly this should already happen in the phase before through the programmers' self control. During the phase at hand metric-tables can get changed too, in order to fit "coding reality". In both cases the loop back to phase "metric guided programming" is taken. If the SE's results and the metric-model meet, the cycle advances to the following step.
The benefit in comparison to the usual review process is: The ME can concentrate on the parts of the product which seem to be more complex and less documented. In order to be able to fine-tune his

measuring tools for the various projects in the software development department, the ME will certainly need an environment which enables him to shape his metrics in a simple way. This is the idea of the measuring tool generator described in part 3 of this paper.
Revision of metrics: The degree of fitness of the PM's qualitative requirements is compared to the actual results of maintaining the quantitative indicators. The border-values and weights of the metrics get adapted. These new metric-tables can be used during step two of a similar project, where, if project constraints are comparable, the previous metric-table can serve as a useful initialization.

Within the whole process of quality-guided development one can regard the so-called quantitative evaluation as one special form of review. To say it in other words: anything that can be automatically measured by SW metrics could be a result of human review, too. But reviewing is costly. So one of the goals of any QS group should be to move criteria control from human reviewing to metric analysis as much as possible (see fig. 2).
Putting the above cycle into actual work requires a tool which can be adapted easily by the metric engineer. It is obvious that adaption is somehow permanent, therefore the client and not only the tool vendor has to be capable to generate new tools. The best place for this tool would be that of a component of some development framework (e.g. a CASE-tool).

3. Independent definition of language and metric: a flexible tool as a necessary condition

Code analyzers and measuring tools are useful for the automation of formal source code evaluation. They are of particular use during the phases of testing and maintenance of ill-documented source code. Traditionally code analyzers have been built one by one including the knowledge of the programming language to be parsed and the code characteristics to be evaluated. This approach is of limited flexibility. If the language to be parsed or the metrics to be measured change, any adaption is difficult: the internal structure of the measuring tool must be known to the person performing the change. The use of parser generators like Lex and Yacc for language definition is a simple way of improvement which we applied here. Within these the functionality of the code analyzer is stored in semantic actions scattered all over the language definition in EBNF.
In case of a change in grammar (e.g. due to extensions) it is quite easy to adapt. However if you want to evaluate code according to different sets of metrics the metric procedures have to be coded by hand into several instances of the grammar representations.

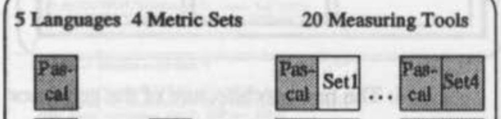
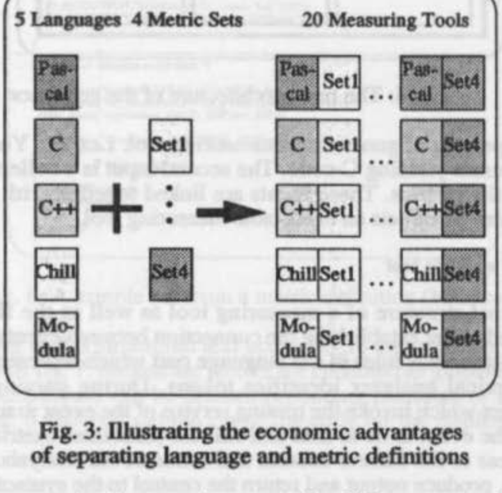



Fig. 3: Illustrating the economic advantages
of separating language and metric definitions

A quality assurance group typically experiments with different sets of metrics over different programming languages. Reuse is desirable in this case.

A clear separation into two independent representations made obviously sense: in one the language is formally defined and in the other the metric algorithm. These two modules communicate via standardized event messages thus making both modules reusable on source and object code level.

3.1 The basic architecture of the tool

If you want to implement measuring tools for N metric sets on M programming languages, M x N tools are needed. Traditionally each of these M x N tools have been coded. Separating language and metric definitions, the effort can be reduced to M + N implementations. Figure 3 illustrates the potential of savings through separation.

Fig. 4 shows the overall structure of a measuring tool: It consists of the three parts language description, event manager and metric procedures. The *language description* consists of a language grammar description and tokens. Standard events are provided there to be triggered at appropriate times during measurement. The analyzing *metric procedures* are operations which compute metric values and generate output. The *event manager* serves as an interface between language and metric definition. It can understand a set of standard events which are generated by the language part and routes them to the appropriate procedure in the metrics part. So the event manager provides two services:

- "*enter_event*" enters a relation of a standard event to a procedure, which shall act upon this event at run-time, into a routing table at initialization time.
- "*event*" routes a particular event via use of the routing table to a metrics procedure at parsing time.

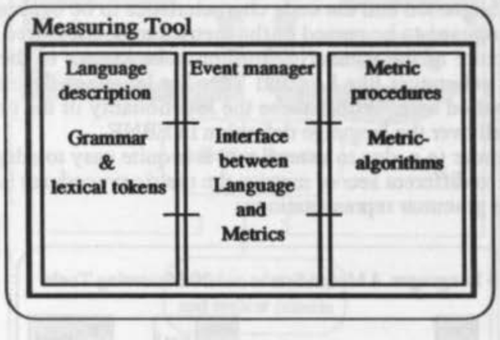


Fig. 4: The basic architecture of the generator

Figure 5 shows the actual process of generating a measuring tool. Lex and Yacc are used for language definition and formal grammar yielding C-code. The second input is a collection of C-functions (see bottom middle) to define the metrics. These inputs are linked together with the fixed C-code of the event manager. Linking them constructs an executable measuring tool.

3.2 Generating and adapting a metric tool

In the following the internal structure of a measuring tool as well as the flow of data and flow of control within are described. After establishing the connection between events and procedures, control is passed on to the EBNF-grammar rules in the language part which represent the syntactic structure of the text input. The lexical analyzer identifies tokens. During parsing the analyzer triggers standardized event messages which invoke the routing service of the event manager.
The event manager uses the event table to find and call the associated metric procedure. Any active metric procedure has access to the current context and status of the analysis. The metric procedures compute the metric values, produce output and return the control to the syntactic parser.

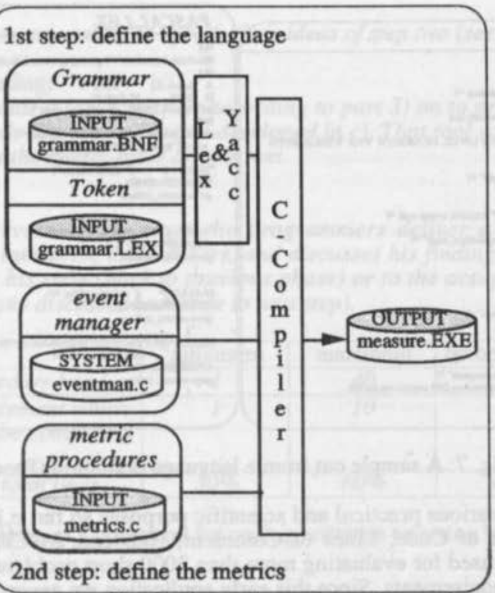


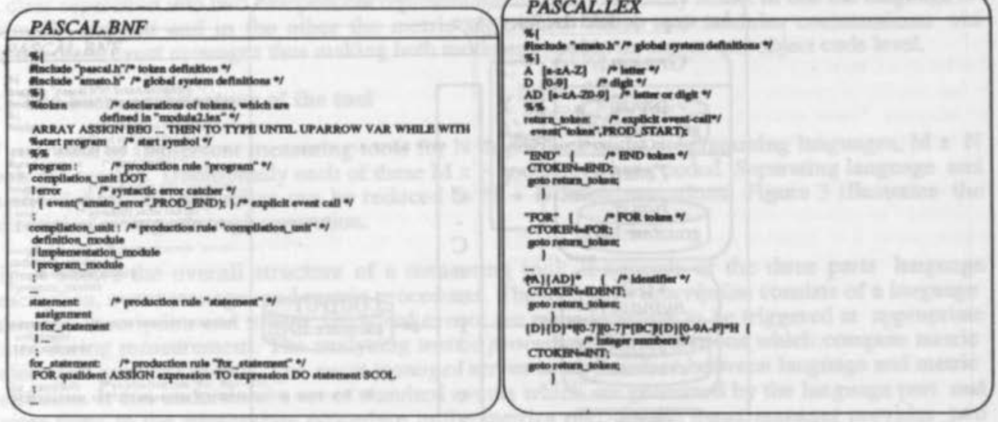
Fig. 5: Generating a measuring tool

Halstead.C

```
#include "amato.h"
int ns=0,n1=0; /* number of statements & unique operators */
int n2=0,hN1=0,hN2=0;
  /* number of unique operands, all operators & all operands */
/* event procedures */
void inc_ns() /* increment the number of statements */
{ ++ns; }
void inc_hN1() /* increment the number of operators */
{ hN1+=n1_tab[CTOKEN]; } /* hN1 gets incremented only if the
                         operator is not yet in the operator list */
...
void epilog() /* write out a summary */
{
  fprintf(COUTPUT,"\nHalsteads N1 counts %d.",hN1);
  fprintf(COUTPUT,"\nModule '%s' counts %d lines.",CTEXT,CLINE);
  ...
}
init()        /* initialize event-table */
{
  enter_event("statement",PROD_START,inc_ns);
  enter_event("for_statement",PROD_START,inc_hN1);
  enter_event("expression",PROD_END,inc_hN1);
  ...
  enter_event("compilation_unit",PROD_END,epilog);
}
main()        /* main program */
{
  COUTPUT=stdout;
  CINPUT=stdin;
  init();
  yyparse();
}
```

Fig. 6: A sample cut from a metric definition (Halstead)

The input to the measuring tool is structured text according to the language definition in source code like the Pascal statements in fig. 7. Measuring results are collected as global data in the event manager. In fact any output event procedure can be defined producing printouts or file-dumps at any desired status during parsing. Fig. 6 shows a sample cut of the code for the implementation of the Halstead complexity metric.



PASCAL.BNF

```
%{
#include "pascal.h"/* token definition */
#include "amato.h"  /* global system definitions */
%}
%token          /* declarations of tokens, which are
               defined in "modula2.lex" */
 ARRAY ASSIGN BEG ... THEN TO TYPE UNTIL UPARROW VAR WHILE WITH
%start program     /* start symbol */
%%
program :          /* production rule "program" */
 compilation_unit DOT
 | error          /* syntactic error catcher */
  { event("amato_error",PROD_END); } /* explicit event call */
 ;
compilation_unit : /* production rule "compilation_unit" */
 definition_module
 | implementation_module
 | program_module
 ;
...
statement:         /* production rule "statement" */
  assignment
  | for_statement
  | ...
  ;
for_statement:     /* production rule "for_statement" */
 FOR qualident ASSIGN expression TO expression DO statement SCOL
 ;
...
```

PASCAL.LEX

```
%{
#include "amato.h" /* global system definitions */
%}
A  [a-zA-Z]      /* letter */
D  [0-9]         /* digit */
AD  [a-zA-Z0-9]   /* letter or digit */
%%
return_token:     /* explicit event-call*/
  event("token",PROD_START);

"END"  {        /* END token */
 CTOKEN=END;
 goto return_token;
    }

"FOR"  {        /* FOR token */
 CTOKEN=FOR;
 goto return_token;
    }
...
{A}{AD}*    {  /* identifier */
 CTOKEN=IDENT;
 goto return_token;
    }
{D}{D}*|[0-7][0-7]*[BC]|{D}[0-9A-F]*H  {
              /* integer numbers */
 CTOKEN=INT;
 goto return_token;
    }
```

Fig. 7: A sample cut from a language definition (Pascal)

We generated tools for various practical and scientific purposes so far: e.g. a Modula-2 parser for a metric set around Lines of Code, Lines of Comment, Halstead, McCabe, Henry & Kafura and Rechenberg, which was used for evaluating more than 2000 short problem programs written by 500 students on 4 problem requirements. Since this early application we generated tools for PASCAL, C, C++ and Chill, implementing many major static code metrics for procedural languages and some for their object oriented extensions. In any case the presented concept proved to be successful with respect to reusability of metric descriptions.

4. Applying the tool within a quality assuring process: an everyday example

The generator's main design principles are flexibility and usability. These are characteristics on a purely technical and theoretical level at first. Dealing with "everyday software development" in industry these criteria come up in a more practical level also. The former has been described in part 3. The latter is illustrated in this chapter: The actual application of the generator covers the steps *selection of metrics* and *metric guided programming* (see grey area in fig. 2). The scenario at hand is a contract of a software developer whose client is conscious about possible maintenance costs. The process described in part 2 can be sketched by the following six steps:

Analysis of goals:

The PM has to plan a specific development project. In the contract it says that in case of high maintenance costs they have to be covered by the supplier. Therefore the main requirement is low maintenance cost over the product's life-span. Another requirement is a guaranteed repair-time of not more than one week in case of errors. The PM can define these requirements as more or less informal quality attributes. His/her message to the quality assurance group will be: Software must be cheap in maintenance, and short in errors that cannot be eliminated quickly.

Goals/questions/metrics:

With a first glance at the PM's goals the ME can see that they are too informal to be tested. They are also too imprecise to be used as guidelines for the production process. After some arguing on the issues of maintainability and "quick elimination of errors that occur" they agree on the criteria: 1) Any procedure has to be well documented. 2) Complex parts have to be documented intensely. They decide for lines of comment as a measure of documentation. To identify "complex parts of code" they take the measures of total complexity according to [21] with respect to one statement .

Selection of metrics:
The ME and PM set up metric-tables to specify their ideas of step two (see table 1):

Metric guided programming:
The metric engineer hands a tool (creation according to part 3) on to programmers, that allows for quantification of code due to the criteria developed in c). That tool will print out warnings, if the standards defined in the metric-table are not met.

Evaluation and rating:
Either in regular intervals or whenever the programmers deliver a piece of code the ME measures the product, interprets the numbers, and discusses his findings with the SE. This may lead to the SE refining his code (back to previous phase) or to the acceptance of the part of the product which is currently discussed (advance to next step).

Measure	minimum	maximum	favorite value	weight
comments at each procedure head	*1*	*40*	*5*	*0.3*
comment lines at each statement which enclose code whose McCabe complexity exceeds 4	*1*	*10*	*3*	*0.4*
ratio comment lines to total lines	*30%*	*60%*	*40%*	*0.3*

Table 1: Sample metric-tables as a constraint for coding activities targeting lower maintenance costs

Revision of metrics:
After completion of the project the PM and the ME meet again to discuss the success of their policy. They recognize that the code of the recent project is actually better documented as it is usually the case in the department. Due to the selective reviews of the ME during the project some major sources of errors as well as of confusion were removed through documentation or redesign. In the basic questions on maintenance cost and time to fix for errors they agree on having a log-book as a documentation of these items and on meeting for the purpose of a review every quarter. The experience gained in this process will be used to lift the quality of retuned metric-tables thus forming a department-wide learning curve.

Though we collected several experiences in applying the tool for different metrics and languages, we are still in an early stage with evaluating the use of the process described in fig. 2. Up to now we proceeded in testing metric guided programming in student projects of 15000-20000 lines of code in size. Groups of 5-8 people developed a software system from the phase of requirement engineering to preparation for maintenance during a whole academic year. The phase of coding took two months within that time span. We focused on simple metrics like the ratio of LoC and total lines, line oriented complexity limits and higher density of comments at procedure heads. Programmers' reaction was ambivalent. In a way they feel too strongly controlled by unreasonable rules. Obviously they have to be involved in the metric design procedure, too, in order to make them understand a code measuring tool not only as a cut to their creativity but as a useful facility as well.

5. Conclusion

Within this paper we outlined an approach for making quantitative evaluation of software systems useful for practitioners. This approach is based on the idea of developing coding rules locally. A process model has been described which assures quality-guided programming by using the techniques of quantitative evaluation. Its implementation affords a highly flexible tool for measuring source code. The internal structure of a tool-generator has been explained and also its application. On the practical level that tool concept provides flexibility and usability whereas on the technical level it provides a clear separation of metric and language definitions.
Nevertheless, a good tool is just one technical condition for practical expediency of software metrics. Matters of organization and psychology are the subsequent "real" problems of the domain.

Bibliography

[1] Adamov R., Baumann P., Literature Review on SW-Metrics, Institut für Informatik der Universität Zürich, Okt.1987

[2] Albrecht A. J., Gaffney J. E., SW-Function, Source Lines of Code, and Development Effort Prediction: A SW-Science Validation, IEEE TSE, Vol. SE-9, No. 6, pp. 639-648., Nov 1983

[3] Basili V., Tayloring the SW process to Project Goals and Environments, In Proc. of the 9th Int. Conf. on SE, ACM, 1987

[4] Berry, R. E., Meekings B.A.E., A Style Analysis of C Programs, CACM, Vol. 28, No. 1, pp. 80-88., Jan 1985

[5] Binder L.H., Poore J.H., Field Experiments With Local Software Quality Metrics, Software - Practice and Experience, Vol. 20(7), p.631-647, 1990

[6] Boehm B.W. Software engineering economics, IEEE TSE, Vol.SE-10,No.1,p.4-21 Jan. 1984

[7] Boehm B.W., Understanding and controlling software costs, IEEE TSE, Vol.14, No.10, p.1462-77, Oct.1988

[8] Briand L.C., Basili V.R., Hetmanski C.J., Providing an Empirical Basis for Optimizing the Verification and Testing Phases of SW Development, Proc. on Int. Symp. on SW Reliability Engineering, North Carolina, USA, Oct 1992

[9] Conte, Dunsmore, Shen, SW Engineering Metrics And Models, Benjamin/Cummings, 1986

[10] Cote V., Bourque P., Oligny S., Rivard N., J., Software metrics: an overview of recent results, Syst. Softw., Vol.8, No.2, March 1988, p.121-31.

[11] Gill G.K., Kemerer C.F., Cyclomatic Complexity Density and Software Maintenance Productivity, IEEE TSE, Vol. 17, No. 12, Dec. 1991, p.1284-1288

[12] Grechenig Th., Biffl St., Taylor your own metrics environment: AMATO - a tool for the metric-engineer, Proc. of Eurometrics 92, Brüssel, p. 287-300, Apr. 1992

[13] Halstead, M.H., Elements of SW-Science, Elsevier North-Holland, 1977

[14] Hausen H. L.,Müllerburg M., Über das Prüfen, Messen und Bewerten von SW. Methoden und Techniken der analytischen SW-Qualitätssicherung., Informatik-Spektrum, Band 10, 1987

[15] Hon III S.E., Assuring SW Quality through Measurements: A Buyer's Perspective, J. Systems SW, 1990, Vol. 13, p.117-130, 1990

[16] Höckel H., Itzfeld W.D., Qualitätsmaße für SW in der Praxis, ONLINE 9/86, Sept. 1986

[17] Ince D., Software Metrics, Measurement For Software Control and Assurance, Editors: Kitchenham B.A., Littlewood B. London, UK: Elsevier Appl. Sci. Publishers 1989, p. 27- 62, 1989

[18] Kafura, D.; Reddy, G.R., The use of software complexity metrics in software maintenance, IEEE TSE, Vol.SE-13, No.13, p.335-43, March 1987

[19] Monitoring software development using metrics, Kitchenham B. A. UK IT 88 Conference Publication; London, UK: Inf. Eng. Directorate 1988, p. 45-8 of xix+618, Conf.:Swansea, July 1988

[20] Kitchenham B. A.; Walker J. G., A quantitative approach to monitoring software development, Software Engin. Journal, Vol.4, No.1, p.2-13., Jan.1989

[21] McCabe, T.J., A Complexity Measure, IEEE TSE, Vol. SE-2, No. 4, pp. 308-320., Dec 1976

[22] McGarry F.E. Using software metrics and measurements to improve software productivity and quality, Proc. of the Comp. Standards Conference 1988, Washington, DC: IEEE Comp. Soc. Press 1988, p. 105 of x+111., 1988

[23] Oman P.W., Cook C.R., Design and Code Traceability Using a PDL Metrics Tool, J. Systems SW, 1990, Vol. 12, p.189-198

[24] Page D.R., Static Code Analysis For COBOL Development: The Advantages of An Automated Programming Support Tool, Unisphere 8, 12: 64-66, Mar 1989

[25] Paulish Dan, Best practices of software metrics, Tutorial 3, European conference on quantitative evaluation of software and systems, Proc. of Eurometrics 92, Brüssel, 1992

[26] Prather, R.E., An Axiomatic Theory of SW Complexity Measure, Computer Journal, Vol. 27, No. 4, pp. 340-347, Nov 1984

[27] Rechenberg P., Ein neues Maß für die softwaretechnische Komplexität von Programmen, Informatik Forschung und Entwicklung 1; p. 26-57, 1986

[28] Redmond J.A., Ah-Chuen R., Software Metrics - A User's Perspective, J. Systems SW, 1990, Vol. 13, p.97-110

[29] Rombach H.D., SW-design metrics for maintenance, Proc. 9th Annu. SE Workshop, NASA Godard, pp. 100-134, Nov. 1984

[30] Rombach H.D.,Basili V.R., Quantitative SW-Qualitätssicherung. Eine Methode zur Definition und Nutzung geeigneter Maße., Informatik-Spektrum, Band 10, 1987

[31] Rombach H.D., Ulery B. T., Establishing a Measurement Based Maintenance Improvement Program: Lessons Learned in the SEL, Proc. of Conf. on SW Maintenance 1989, Miami FL, p.50-57, October 1989

[32] Samadzadeh M.H., Nandakumar K., A Study of Software Metrics, J. Systems Software, 16; p. 229-234, 1991

[33] Schneidewind N.F., Methodology For Validating Software Metrics, IEEE TSE, Vol. 18, No. 5, May 1992, p.410-422

[34] Sherif, Y.S.; Ng, E.; Steinbacher, Computer software development: quality attributes, measurements, and metrics, J. Naval Research Logistics, Vol.35, No.3, p.425-36, June 1988

[35] Siegel Stan, Why we need checks and balances to assure quality, IEEE Software, Jan.1992

[36] Whale G., SW Metrics and Plagiarism Detection, J. Systems SW, Vol. 13, p.131-138, 1990

[37] Zuse H., Bollmann P., Reply to: Erhard Konrad: Software Metrics, Measurement Theory, and Viewpoints - Critical Remarks on a New Approach, ACM SIGPLAN Notices, Vol. 26, No. 5, May 1991, p.27-36