A NOTION OF RULE-BASED SOFTWARE QUALITY ENGINEERING *
Hans-Ludwig Hausen
GMD, Schloss Birlinghoven, D-5205 St. Augustin 1, hausen@dbngmd21.bitnet

Abstract

Engineering of software quality is defined by rules for methods, tools, evaluation, measurement and assessment. The product model used requires at least three layers: application concept or requirements specification, data processing concept or system specification, and realization or programs. For a complete evaluation, assessment and certification of a software system one has to examine and assess all its components described by these layers.

Quality is decomposed into evaluation factors and quality factors. Evaluation factors define quality factors; metrics for functional and non-functional requirements are the terms used to define evaluation factors. All these are dependent upon quality objectives and goals.

Methods and tools for evaluation, measurement and assessment are also imposed by objectives as they are connected to the factors, methods and tools. Each problem domain is represented by a set of rules. Problem domain dependencies are also expressed by rule sets. This might provide the basis to define a framework for learner controlled instruction on software quality engineering. Which is subject for further work.

1 Introduction

In this preface we briefly describe software development, evaluation, measurement, and assessment in order to ease the understanding of the discussion in the later sections. We explain the terms used and their meanings. The present section, therefore, is written as a short state of the art statement reflecting a particular view on software engineering, its methods and its tools. [scope91] provides a more detailed discussion.

1.1 Software Layers and Views

At the beginning, right after a feasibility study, we have collected a set of activities and documents (to be) performed and handled in an application department. In the modeling of the application, where we answer the question: "What do we want to use the system for?", those activities and documents must be identified, which have to be performed and represented by functions and data elements of a data processing system. Assigning functions to activities and data to documents, we obtain a function model and a data model of the software system to be developed. By this, we solved the problem: "What is to be performed on the computer?". In order to define precisely what kind of operation is to be performed the definition of the states to be assumed is required for both, the application concept and the data processing concept. The former is defined in terms of activities and documents, the later is described by instances of data and functions from data processing concept.

Executable computer programs are those parts of a software system, which implement the data processing concept (also called system specification). The programs are the answer to the question: "How has the computer to work in order to achieve the What and the What for requirements?". Programming language elements, such as procedures or modules, are used to code functions; structured variables are taken to represent data structures. Using a requirements specification language for the application concept, a specification language for the data processing concept and a high level programming language for the programming we obtain a three layer description of the functionalities of the system to be engineered. Depending on the type of languages chosen we obtain a textual or a graphical representation, or a mixture of both.

Additionally, a documentation is produced, which comprises the formal descriptions (application specification, data processing specification, programs) as well as a natural-language description of tasks, requirements, conditions, approaches to solution, etc., i.e. a verbal explanation of all problem dimensions of the particular system. Within an optimal software engineering process both the formal and the informal descriptions are produced in parallel. This ensures that decisions taken are documented and that alternative solutions have been considered.

---

*The work reported has been founded partly by the CEC, Esprit-II Project SCOPE, No. 2181, Software Assessment and Certification Programme Europe
Figure 1: Objects and Layers of Software Development

These three layers represent the most significant descriptions of software. The main tasks leading to those layers are executed in sequence, but some tasks have to be performed in parallel. Often tasks are grouped according to the type of individuals (such as designer, programmer, or tester) working on them. Task groups are considered as views on the software engineering process. A view is to be described by all actions and items handled with respect to a selected set of goals and objectives. In most process models on can find a construction view, a validation view, and a management view. A view defines the distinguished responsibilities within the engineering process. A more detailed discussion on product models and processes models and their implementation is to be found in [Haus87x].

1.2 Quality Engineering

As in other engineering software engineers' have to try to avoid the well-known problem of undesirable developments recognised too late. Simultaneous quality evaluation and assessment of intermediate and final result is an appropriate way to handle this. Effective evaluation and assessment requires the appropriate use of validation methods, such as: inspection, walkthrough (a.o.), statistical analysis, static analysis, metrics, dynamic analysis, testing, test quality, symbolic execution, symbolic evaluation, formal verification, quality modeling, productivity modeling, security analysis, reliability, project and product management. Non automated evaluation, called inspection (or walkthrough, etc.) is one of the most effective, but also most expensive technique, because of the costs of the inspectors. Its instrumentation by information systems (for missing items retrieval for example) and conferencing systems is suitable and might be enhanced by grammar-directed incremental analysis of natural language documentations.

Static analysis covers all compiler-based evaluation techniques (e.g. scanning, parsing and static semantics analysis). This includes also the measurement of non-functional attributes (such as number of pathes, statements, calls, etc.). A compiler-oriented language definition is required for each language applied, even for the documentation language. The use of other languages must be refused, at least form the quality engineer's point of view. Tools for static analysis should be generated using compiler-complier.

The evaluation of structural complexity as a key quality attribute today is defined for control flow, data flow, state transitions, calling structures, as well as for text structures. Its instrumentation might be done by specific tools, but it should be included into the compiler. At present, there are to many structural complexity metrics proposed, whose validity has not been proven. From a mathematical point of view, some of them would benefit if a well-defined notion would be made available. On the other hand, logical complexity (computation complexity) testing is well defined in the theory of algorithms. But, unfortunately, it is not considered as important to quality engineering as it deserves.

The most wide-spread evaluation technique is testing, which comprises black box and glass box testing. The former investigates the actual input-output relation of an executable code with respect to its specified version. At present data for black box test are selected based on experience or special boundary identifications, or are derived from specifications. Quality then is defined as achieved coverage of the input domain or coverage of checked functionalities. Glass box testing considers the internal structure of an executable code. From a code (for a program, a specification, or an application) we obtain by abstraction data flow, control flow or state transition. On a more detailed level we get also a structural decomposition of the expressions or predicates used in assignations or case structures. Strategies and data for glass box testing are defined by means
of language-element-specific quality criteria for flow of control testing (a% expression coverage, b% predicate coverage, ... x% path coverage, y% instruction coverage, z% module coverage, ..., and hybrids), data flow testing (r% data relations of type t, s% value definitions, t% value referencing, ..., and hybrids) or state testing (m% expression values, n% predicate values, ..., and hybrids). Testing became a mature technique for (well-structured) sequential programs, but for others, some severe theoretical problems still have to be investigated.

Another wide-spread technique is formal verification, which is most important to evaluation of reliable and secure software. In a verification the correctness (i.e. computation of required output and termination) is shown by means of a formal proof, using (a particular) mathematical logic. A lot of theorem proving methods have been successfully applied to check correctness. Unfortunately the techniques to construct proofs are not mature for industrial application.

The assessment of a product (leading to a quality statement) or a process (leading to a productivity statement) requires an evaluation of the actual extension of both the functional and the non-functional attributes of a software system, which are defined and measured by metrics for product attributes or process items. From literature one can get a great number of so-called quality or productivity models, each describing the set of important factors, attributes, metrics, and their interrelation. Since each software is unique, the use of one particular quality or productivity model for all kind of software is inappropriate. We, therefore, suggest a generic model which is to be tailored to project and product characteristics, with respect to goals and objectives of a particular engineering process.

In this context assessment is considered as testing whether required or expected conditions have been fulfilled for the product, the process, or both. For an assessment we then compare the set of given factor values (target) against the set of factor values which are present (actual state obtained by examination and measurement). We obtain ranking statements, such as program X is more efficient than program Y or the actual structural decomposition of an application concept is more complex than its target. Our approach to quality assessment requires both specification of quality requirements (target or required quality graph) in terms of factors and metrics, and evaluation of their actual extensions (actual quality graph), where both graphs are of identical structure.

A list of the most interesting factors might include: correctness, reliability, integrity, maintainability, flexibility, portability, reusability, testability, usability, or efficiency. As an example for quality modeling one might want to consider the following:

- workmanship
  objectives: Compliance with good engineering practice?
  or: Is the right system built right?
  related quality criteria: complexity, consistency, expandability, generality, modularity, revisability, self documentation, simplicity

- correctness
  objectives: Implements the product the specifications?
  or: Is the right system implemented right?
  related quality criteria: completeness, consistency, traceability

- reliability
  objectives: Runs the system reliable under specified conditions?
  or: Is the right system applied right?
  related quality criteria: accuracy, (consistency), fault tolerance, (modularity), (simplicity)

- efficiency
  objectives: Good use of available resources in space and time?
  or: Is the right system performing at right costs?
  related quality criteria: correctness (functional and operational), complexity (structural and logical), performance, storage usage

Complexity is one of the most often used "metric", it might be measured by counting occurrence or frequency of sub-elements (e.g. subroutines), nested elements, defined elements, referred elements, predicate variables or other variables. For connectivity, another frequently selected "metric", one might be interested in module dependence (to be measured by the counting occurrence or frequency of: modules, module invocations or passed data items) or program dependence (to be determined by the counting occurrence or frequency of: functions or data used in more than one program, file accesses, data base accesses, operating system accesses or user interface invocations). To assess a specification we need to measure: specification text (i.e. the string of specification) or the static structure of the specification expression (i.e. the structure of the specification functions). For a program we want to measure program text, control flow graph, data flow graph, storage space usage or execution time consumption. Program text is measured by: number of chars, number of words, number of sentences, number of chapters, number of variables and types, number of operators and control structures, frequency of variables and types or frequency of operators and control structures. For all kind of graphs we are able to measure them by counting occurrence or frequency of edges, nodes, subgraphs, connected subgraphs, strong
connected subgraphs, maximal connected subgraphs or
nesting of subgraphs.

What do we need for effective quality engineering?
Within the approach proposed one needs to define three
layers of the product, quality objectives, quality factors,
metrics, and methods and tools to evaluate the product.
The question know is, what is the appropriate way of
defining these. Our solution is to use rules of the form:

\[
\text{IF} \quad \text{(sub-) domain } a, \text{ (sub-) domain } b, ..., \text{ (sub-) domain } s \text{ is specified}
\]

\[
\text{THEN} \quad \text{domain } D \text{ is defined}
\]

\[
\text{ENVIRONMENT} \quad \text{environment part identifies}
\]

\[
\text{EXPLICATION} \quad \text{application area i.e. scope of}
\]

\[
\text{EXPLANATION} \quad \text{the rule}
\]

\[
\text{EXPLANATION} \quad \text{justifies definition, has to be provided by}
\]

\[
\text{EXPLANATION} \quad \text{the designer of the rule}
\]

\[
\text{EXPLANATION} \quad \text{part}
\]

\[
\text{EXPLANATION} \quad \text{justifies application, has to}
\]

\[
\text{EXPLANATION} \quad \text{be recorded by/for the user of}
\]

\[
\text{EXPLANATION} \quad \text{the rule}
\]

\[
\text{EXPLANATION} \quad \text{to model each particular problem domain and the}
\]

\[
\text{EXPLANATION} \quad \text{interrelation of them.}
\]

1.3 A Formal Notion of Quality

As we consider each software system (or process) as
a unique item we define quality (or productivity) by
a generic framework, i.e. by an acyclic decomposition
graph, whose intermediate nodes are quality (or pro-
ductivity) factors and whose final nodes are atomic at-
tributes of the product (or process), the later measured
by a well-defined metric (called evaluation factor). As
a metric is a homomorph mapping of the object to be
measured onto a nominal, ordinal, interval or rational
scale, we include all kind of evaluations from mappings
onto real numbers down to checklists or enumeration.

In order to have a precise notion of quality we intro-
duce the following formulas: We start with:

\[
\text{actual extension of evaluation factors}
\]

\[
\text{actual extension quality or productivity factors}
\]

\[
\text{where}
\]

\[
\text{target extension of evaluation factors}
\]

\[
\text{target extension of quality or productivity factors}
\]

\[
\text{WE GET AN actual system value vector AS:}
\]

\[
\text{WE START WITH: actual system value tuple AS:}
\]

\[
\text{WE GET AN actual system value vector AS:}
\]

\[
\text{AS ALL ASVI AND RSVI ARE OBTAINED WE GET A}
\]

\[
\text{level-s system value:}
\]

\[
\text{2 Problem Domains}
\]

Within this section we discuss the rule-based handling
of methods, tools, quality, and objective, i.e. the goals
and objectives of quality engineering. For each prob-
lem domain it will be shown how rules can be used to
eengineer its general task as well as all subtasks.

2.1 Quality Methods and Tools

On of the key problems are the handling of methods
and tools to be used in the evaluation, measurement or
assessment of software. We consider methods and tools
as a basic dichotomy since the methods define what actions we have to do whereas the tools dictate how we have to perform them. On the other we can take the dichotomy as an advantage, because it permits us to model each domain separately. We get

\[
\begin{align*}
&\text{IF } (\text{sub-}) \text{ method(s)} \\
&\text{THEN } \text{method} \\
&\text{IF } (\text{sub-}) \text{ tool predicates} \\
&\text{THEN } \text{tool predicate} \\
&\text{and for the interrelation:} \\
&\text{IF } \text{tool predicate} \\
&\text{THEN } \text{method predicate}
\end{align*}
\]

In order to be able to express preferences we extended the rule scheme by ranks. This is shown by the following example:

\[
\begin{align*}
&\text{IF } \text{rank } \in \{0 \ldots 1\} \\
&\text{THEN } \text{ENV.} \\
&\text{EXPLC.} \\
&\text{EXPLN.}
\end{align*}
\]

Using this rule the evaluation techniques inspection, testing, verification might be ranked for a validation to be appropriate in the environments SM, LM, VM or RM. By a rule like:

\[
\begin{align*}
&\text{IF } \text{rank } \in \{0 \ldots 1\} \\
&\text{THEN } \text{LINE EDITOR} \\
&\text{SCREEN EDITOR} \\
&\text{GRAPH EDITOR}
\end{align*}
\]

we rank the selection of editors for a generic tool EDITOR, that is applicable within the operating systems OS1, OS2 or OS3. Other rules define the methods and tools for code instrumentation and measurement by introducing new attributes to the grammar that is used to generate the compiler by compiler-compiler tools such as lex, yacc or awk.

### 2.2 Quality Modeling

As introduced above quality (or productivity) is described by an acyclic graph which decomposes high-level factors into (low-level) metrics. Such structured definitions are represented by rules of the form:

\[
\begin{align*}
&\text{IF } \text{is - evaluation - } \text{factor}(e_1, \ldots, e_n) \\
&\text{THEN } \text{is - } \text{quality}(f_1, \ldots, f_n, e_1, \ldots, e_n) \\
&\text{ENV.} \\
&\text{EXPLC.} \\
&\text{EXPLN.}
\end{align*}
\]

But we can also define our formulas by rules, which leads us to:

\[
\begin{align*}
&\text{IF } (f_{h,i}(a_{h,i}, e_{f1}), \ldots, f_{n,i}(a_{h,n}, e_{f(n,i)})) \\
&\text{THEN } f_i \\
&\text{and finally to} \\
&\text{IF } (f_{h,i}(a_{h,i}, e_{f1}), \ldots, f_{n,i}(a_{h,n}, e_{f(n,i)})) \\
&\text{and } (f_{i1}(a_{i1}, f_i), \ldots, f_{i,g}(a_{i,g}, f_{i,g})) \\
&\text{and } (f_{g1}(a_{g1}, f_{g1}), \ldots, f_{g,l}(a_{g,l}, e_{f(g,l)})) \\
&\text{THEN } Q
\end{align*}
\]

Since metrics are (partially) structured we are also able to define them by rules, as we show in the next example for the well-known cyclomatic complexity of control-flow graphs:

\[
\begin{align*}
&\text{IF } \text{is - number - of - nodes}(n, cfg) \\
&\text{is - number - of - edges}(e, cfg) \\
&\text{is - number - of - graphs}(p, cfg) \\
&\text{is - value - of}(c; (e - n + 2p)) \\
&\text{THEN } \text{is - cyclomatic - number - of}(c, cfg)
\end{align*}
\]

As shown in this example every aspect we need to know can be represented, and subsequently retrieved, in the rule sets.

### 2.3 Quality Objectives

In order to make a particular notion of quality clear or acceptable we have to explain the objective to develop that notion. One way of presenting a objective is to report sub-objectives and goals, which impacted the particular structuring of the quality model, the selection of the evaluation or assessment methods, and the tools to actually perform the examination. We recommend to apply the well known Roman principle of divide et impera (divide and conquer) to the objectives also. As result we get a structured objective, defined in terms of sub-objectives or goals. In other words, we need to develop a acyclic graph, that decomposes our objective such that we can identify impacts on quality, methods and tools. This leads to objective rules of the form:

\[
\begin{align*}
&\text{IF } \text{set of atomic objectives} \\
&\text{and } \text{set of subsub objectives} \\
&\text{and } \text{set sub objective} \\
&\text{THEN } \text{objective}
\end{align*}
\]

For some quality models we can find rules like

\[
\begin{align*}
&\text{IF } \text{maintainability} \\
&\text{and } \text{testability} \\
&\text{and } \text{flexibility can be achieved} \\
&\text{and } \text{reusability can be achieved} \\
&\text{THEN } \text{objective}
\end{align*}
\]

This example demonstrates that we are now able to describe an objective by the same type of rules as we used to define quality, methods, tools and their interrelation.
2.4 Quality Application

Software has to be monitored during its engineering process. Monitoring on the other hand needs to be focussed on particular attributes, such as volume or effort. In most cases one can reduce the set of attributes to be observed to just one if the objectives are clear and well understood. In order to show how our approach is applicable to these problems, the discussion of this subsection is focussed on volume and effort, since effort is a function of volume as effort might be a function of correctness.

Following our layered structuring of the software we define volume as:

\[
\text{is-volume} = \text{is-volumeprogram}(\text{prgv};\text{sw-sys})
\]

\[
\text{is-volume} = \text{is-composed-from}(\text{v};\text{prgv},dpcv,apcv)
\]

\[
\text{IF} \quad \text{is-volume}(\text{v};(\text{apcv},dpcv,prgv),\text{sw-sys})
\]

\[
\text{THEN} \quad \text{is-volume}(\text{v};(\text{apcv},dpcv,prgv),\text{sw-sys})
\]

\[
\text{is-composed-from}(\text{v};\text{prgv},dpcv,apcv)
\]

Then we get an effort rule scheme of the form:

\[
\text{is-effort}_{ap-concept} = \rho_{ap-c}(\sigma_{ap-c}(\tau_{ap-c}(
\begin{align*}
&f_{c,a}(q_{a} \times \text{volume}_{activities}) \text{ plus } \\
&f_{c,b}(q_{b} \times \text{volume}_{activities})) \\
&f_{c,c}(q_{c} \times \text{volume}_{functions}) \text{ plus } \\
&f_{c,d}(q_{d} \times \text{volume}_{data}))
\end{align*}
\))
\]

\[
\text{is-effort}_{dp-concept} = \rho_{dp-c}(\sigma_{dp-c}(\tau_{dp-c}(
\begin{align*}
&f_{d,p}(q_{p} \times \text{volume}_{procedures}) \text{ plus } \\
&f_{d,q}(q_{q} \times \text{volume}_{variables})
\end{align*}
\))
\]

\[
\text{is-effort}_{program} = \rho_{program}(\sigma_{program}(\tau_{program}(
\begin{align*}
&f_{p,a}(q_{a} \times \text{volume}_{activities}) \text{ plus } \\
&f_{p,b}(q_{b} \times \text{volume}_{activities}))
\end{align*}
\))
\]

Using these formulas we can monitor the product during its entire life-time. The cost-driving functions might be adjusted during or after a project is finished. The total effort is defined as:

\[
\text{is-effort}_{ap-concept} = \rho_{ap-E}(\sigma_{ap-E}(\tau_{ap-E}(
\begin{align*}
&f_{a,p}(q_{p} \times \text{volume}_{activities}) \text{ plus } \\
&f_{a,b}(q_{b} \times \text{volume}_{activities}))
\end{align*}
\))
\]

\[
\text{is-effort}_{dp-concept} = \rho_{dp-E}(\sigma_{dp-E}(\tau_{dp-E}(
\begin{align*}
&f_{d,p}(q_{p} \times \text{volume}_{procedures}) \text{ plus } \\
&f_{d,q}(q_{q} \times \text{volume}_{variables})
\end{align*}
\))
\]

\[
\text{is-effort}_{program} = \rho_{program}(\sigma_{program}(\tau_{program}(
\begin{align*}
&f_{p,a}(q_{a} \times \text{volume}_{activities}) \text{ plus } \\
&f_{p,b}(q_{b} \times \text{volume}_{activities}))
\end{align*}
\))
\]

As we have shown for volume and cost, we can define the interrelation of product attributes by our rule-based, generic notion of quality if we introduce rules for the cost triggers. The quality attributes are already defined and also the methods and tools. The next problem to be solved is the application of all these.

3 Rule Applications

The remaining question is: "How do we use the rule sets?" As we have described each problem domain by a set of appropriate rules we have to define the interaction of the rule sets. A second question is: "How to interpret the rules?" Both will be discussed in the succeeding subsections.

3.1 Rule Set Interrelations

The task is to define the interaction of objectives, quality factors, methods and tools. In manual engineering it is a project manager who has to instrument a project. S/He has to identify factors of quality or productivity, methods and tools to be used in the project in order to achieve the goals and objectives. As each problem domain (goals, factors, methods, tools) is defined and structured by appropriate rules, their interrelation are also described by rules.

We provide two alternative ways of rule set interactions. In approach one a set of interaction rules defines (or instruments) objectives in one step, where each interaction rule implements an objective by the factors, methods and tools identified as being relevant. In the alternative approach we are going form objectives to factors, from there to methods, and finally to
the tools. In this case we have three connected sets of interrelation rules.

### One Step Interrelations

<table>
<thead>
<tr>
<th>IF</th>
<th>THEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>(factor and method and tool)</td>
<td>EXPLIC author justification</td>
</tr>
<tr>
<td>or</td>
<td>ENV parameter</td>
</tr>
<tr>
<td>then</td>
<td>RANK rank</td>
</tr>
</tbody>
</table>

### Three Step Interrelations

<table>
<thead>
<tr>
<th>IF</th>
<th>THEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>(factor and method and tool)</td>
<td>EXPLIC author justification</td>
</tr>
<tr>
<td>or</td>
<td>ENV parameter</td>
</tr>
<tr>
<td>then</td>
<td>RANK rank</td>
</tr>
</tbody>
</table>

3.2 Rule Interpretations

Putting the domain rule sets and the interaction rules together one might assume that we obtain a large set of rules without any structure. But this is not true, because we have ranks, and parameters within our rules. The first allow us to structure the selection of rules in a top-down manner. Parameters, on the other hand, allows to select only those rules which are valid within a particular environment. As a consequence, we have to visit only those rules which are "relevant" for a particular question.

How do we process the relevant rules? In general we have three possibilities to evaluate our rules. First we might want to go top-down in each problem domain. Bottom-up might be a second approach. But how can we decide, for a given question, on what is bottom or what is top in our rule set? The best strategy would be a guided back-tracking, first within the problem domains and secondly via the interrelation rules. Unfortunately we don't have reflective rules, which would allow us to modify rules during interpretations. Unification is employed, for example, to handle the "similarity" of rules set interactions, since all kind of interactions of two particular rule sets might be necessary for any other pair of rule sets.

The approach to rule interpretations proposed might be viewed as complex. But our implementation of rules for software validation, testing and measurement in Prolog shows that the approach is easy to implement and leads to efficient rule interpretations.

4 Further Work and Conclusions

We have solved the problem of how to define quality engineering and how to instrument it. The approach proposed uses rules for handling both definition of problem domains and domain interactions. But what else can we use the approach for? One interesting field in our view is to instruct engineers how to do a task. The approach for this is briefly described in the following subsection.

4.1 Further Work: Quality Instruction

The most useful "notion" is the one a practitioner can adopt to her/his particular needs. Within the quality engineering approach proposed one might want to have two alternative forms of instruction. In one ap-
proach we, as a learner, want to have direct access to the subject following our individual instruction goals and strategies. An instruction strategy might work top-down, bottom-up or intertwined within each problem domain. Our second approach considers instruction goals, instruction strategies, learner models, learner environments, quality and productivity goals and objectives, quality and productivity model, assessment methods as well as assessment tools as the problem domains of instruction, which have to be related. The instruction problem domains as well as their interrelation are described by a set of rules.

Given these rules a learner has the possibility to adopt and customize an instruction procedure defined by an instructor to his particular needs and requirements. Putting the pre-defined instruction rules into the background s/he can change, enhance, introduce rules in each problem domain. A learner can develop individual instruction objectives and strategy as well as their implementation. Both an instructor and a learner might define guided tours through the problem domains, direct access to particular sub-problems, or ad-hoc touring of software quality engineering.

4.2 Summary
Software quality engineering techniques such as inspection, testing, verification, symbolic execution, measurement and assessment are considered as the tasks to be instrumented. Rules are used to model the knowledge of the particular problem domains as well as their interdependencies and interrelations. The notion of software quality engineering developed can be implemented by Prolog predicates, as experiments for software validation and measurement have shown. Some further work will be focussed on how to use the approach to instruct practitioners. Other work will use the approach to develop a European software assessment and certification scheme.

Figure 4: The Quality Model of McCall

References

[Note ] Referenced literature and further readings


