Realizing a Software Design Measurement Tool: Concepts and Results

Christof Ebert[1]

Alcatel SEL, Communication Systems, Stuttgart, Germany

Measuring the aspects of software complexity during the design phase strongly helps to improve the software development by providing control and management instruments early in the life cycle. A method to integrate software design metrics into commercial software development environments is introduced. The main concepts of the environment including the underlying model and its flexibility are explained, its use and operation is discussed, and some results of the implementation and its application to industrial projects are given with examples.

Keywords: CASE, CASE tool, complexity, design metrics, metrics, system design support.

1. Introduction

A science is as mature as its measurement tools.
Louis Pasteur

Current research in the area of software measurement is directed towards using quantitative techniques earlier in the life cycle [Evanco and Lacovara, 1994; Card and Glass, 1990]. Especially the commercial users of software development environments are highly interested in an early detection of change-prone and fault-prone parts of the system because the costs of removal increase exponentially [Stark et al, 1994]. Management of the design process requires a better understanding of design decisions and their relations to all phases of the process. For this purpose we have developed a framework of design decisions and related design metrics that are based on a formal design description. As a matter of fact, our application of design metrics to a real world environment (telecommunications) helped to improve the designs.

Although measurement theory recommends defining '*measures*' as the methods of combining any input with a number with respect to order, scale, and range, while '*metrics*' are certain methodologies of using, applying, or combining different measures, we will follow current literature that emphasizes the use of 'metrics' for both contents. In the field of software metrics we distinguish between process metrics and product metrics [Melton et al, 1990; Fenton, 1991]. Examples for *process metrics* are given in [Stark et al, 1994], classic examples for *product metrics* are described in [Selby and Basili, 1991; Zuse, 1991].

Software metrics that are used to create quality models can be derived from processes or products generated during the life cycle (Fig. 1, horizontal arrows). Quality models, thus, are generated by the combination and statistical analysis of product metrics (e.g. complexity metrics) and product or process attributes (e.g. quality characteristics, effort, etc.) [Stark et al, 1994; Fenton, 1991; Selby and Basili, 1991]. These models are evaluated by applying and comparing exactly those invariant figures they are intended to predict, for example distinct process metrics (e.g. effort, fault rate, number of changes since the project started, etc.). Iterative repetition of this process can refine the quality models hence allowing their use as pre-

[1] This research project has been done while the author was with the University of Stuttgart. He was supported by the German National Science Foundation (DFG) under grant D5-La 297/16

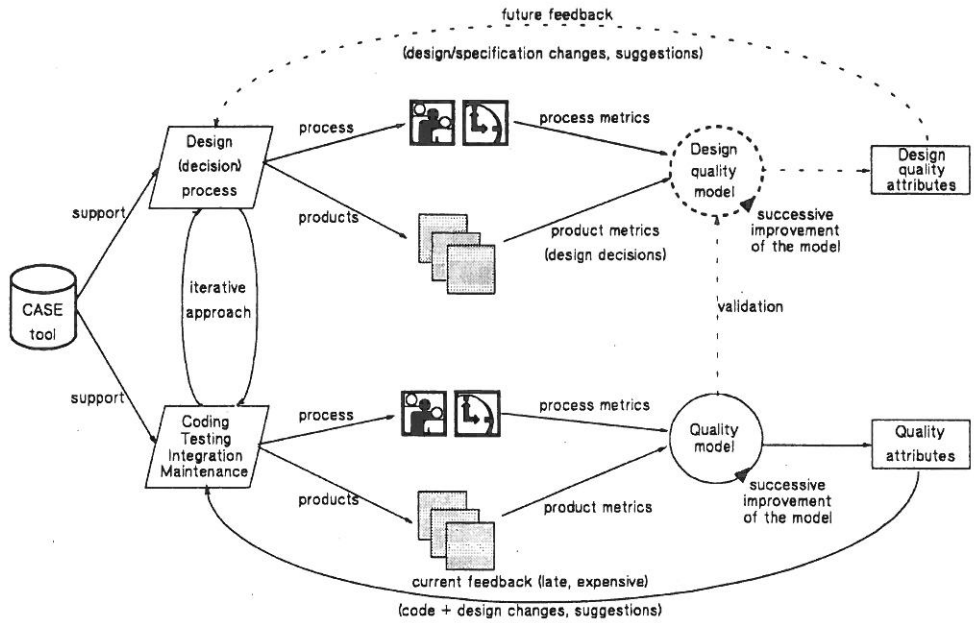

Fig. 1. Metric-based feedback cycles during the development process

dictors for similar environments and projects (circles in Fig. 1). While currently applied quality models for the software development process are primarily focusing on product metrics of the source code [Stark et al, 1994; Selby and Basili, 1991; Porter and Selby, 1990], future control mechanisms should be based on design metrics [Card and Glass, 1990; Al-Janabi and Aspinwall, 1993; Ebert and Riegg, 1991]. The obviously shorter feedback cycles (dotted line in Fig. 1) permit a direct design improvement without waiting for the source code.

Certainly, early complexity estimates are needed as indicators for potentially troublesome components. They could also form a basis for managerial instruments, such as effort or cost estimation and resource planning as soon as possible in a project. An instrument to impose order and structure is classification. Thus, one of the goals of this paper is to show that the classification of all possible design decisions prior to measuring some special attributes that might be related to the design quality, or not, is a more intuitive approach. For the purpose of selecting appropriate metrics we used the classification scheme which ensured that different factors or aspects of complexity were equally considered. One of the earliest studies of design complexity and its factors was presented by David Card [Card and Glass, 1990]. He distinguished design complexity in *functional complexity* resulting from the original requirements (i.e. problem-dependent complexity that usually cannot be controlled by the designer), *system complexity* which includes structural and data complexity (i.e. introduced and controlled by the system designer), and *procedural complexity* (i.e. introduced and controlled by the unit designer).

Among the few frameworks for integrating design metrics is one that was developed at *MITRE* in the USA [Evanco and Lacovara, 1994]. The goal of that project was to provide early indicators for software quality. Unfortunately, only a design metrics *model* is described, yet the integration part does not provide any insight in how to integrate these metrics. The *ESPRIT*-backed *REQUEST* project deals with an automated quality management system that supports quality management during the complete software life-cycle. Another *ESPRIT* project, *COSMOS*, is focusing on metric sets and tool workbenches for real-time systems. Due to the lack of tool support during earlier phases, most other projects that are concerned with analyzing complexity factors and quality concentrate merely on source code [Selby and Basili, 1991; Porter and Selby, 1990]; few deal with detailed design [Card and Glass, 1990; Al-Janabi and Aspinwall, 1993]. In many projects results were either too theoretic or they could not be generalized due to the lack of adequate tool support

or projects to be evaluated. Several reasons can be identified:

- insufficient empirical data;
- isolated tool environments;
- no connection to the development process;
- mere mathematical approach.

Complexity, to our point of view, is a multidimensional attribute such as quality and therefore it consists of different factors. To put it simply, it is insufficient to measure just length or cyclomatic complexity because software products — of all phases of the life-cycle — incorporate other factors of complexity, e.g. functional or data complexity that need to be considered, too. If progress is to be reported during the course of the project, monitoring and tracing comparable complexity metrics for different products is advisable. Comparability of the complexity metrics must be ensured, that is they should measure the same factors of complexity.

Chapter 2 briefly introduces the underlying classification model. Integration of design metrics in a specific CASE environment is described in chapter 3. Chapter 4 provides experience with the measurement tool system and the CASE environment in industrial projects. Finally, chapter 5 summarizes the basic ideas and results of this article.

2. Theory meets practice: Classifying what might be measurable

Conventional wisdom says that you cannot measure things that are neither reproducible nor stable. Therefore, it makes no sense defining measurement programs on a process that is not defined. An answer to the problem might be to start a measurement program upon the new software that is to be developed by the organization. On the other hand, new projects usually take several years to finish their complete life cycle, so it would take the same time to accumulate useful data. It is thus advisable to start the measurement program with already finished projects and collect as much data as available. Some metrics that can be collected from all products are size, complexity, cost, effort, and defects. Data sets from several projects can be ordered according to the distinct quality or productivity aspects that are often well-known among development and sales people. Such external metrics include the number of customer requests, the number of faults reported, the customer satisfaction with the product, or the maintainability in terms of maintenance staff reports.

By applying basic statistical techniques it is possible to get a measurement baseline with almost no effort. These preliminary data sets can function as a benchmark for projects under development, thus improving the data and the measurement collecting process. The described approach has yet another advantage: it is inexpensive *at the beginning*. To tell the truth, implementing a useful measurement program that really controls the process, and hence costs associated with the development, is expensive. David Card and others mentioned additional costs in the range of 5% of total project costs [Card and Glass, 1990].

Design schemes are as different as the design approaches, starting with rapid prototyping (not to mention hacking) up to the very formal methods of a recently graduated Master of Software Engineering. Software design process is the production of a description of an implementation, from which source code can be developed, out of a functional specification and a set of nonfunctional constraints. A *design decision* is any decision concerning the design made by the design staff while producing that description [Rugaber et al, 1990]. *The software design process* is the development of partial solutions for the original problem stated in the requirements specification [Guindon, 1990]. These partial solutions are derived on different levels of abstraction. Each individual solution is a set of design decisions. Different solutions, such as centralized or embedded controllers in an automation project, need to be evaluated against each other and against the requirements. Selection of a distinct solution out of several possible and feasible solutions, for example, is a design decision. *Design metrics* serve as tools that quantify different design decisions in order to support the design process by providing design models. These *design models* are built on former projects and experiences and combine metrics of design decisions with a framework of rules (e.g. limits for metrics, appropriate ranges, statistical background etc.).

To describe it more informally, during the design process distinct decisions are derived from the complete set of different applicable designs in order to fulfill certain requirements. The decisions of what aspects to choose are highly dependent on the concept of modeling the solution (e.g. requirements analysis, system design, design methodology). Depending on these different models for designing the system, several sets of design decisions that form individual solutions for the same problem are possible. Hence, design metrics function as a decision support tool when used and applied correctly, allowing to determine, qualify, and even compare the results of design decisions. For example, a specific design measurement tool that can analyze the size of data structures can be used in this sense by supporting the optimal separation of data objects according to the desired functionality, application, and programming environment [Card and Glass, 1990]. Fig. 1 shows this form of design decision support by using design metrics. While transforming functional specifications and nonfunctional constraints into a software design, a set of distinct design decisions is selected out of all potential design decisions as provided by the supporting CASE-tool. This set of selected design decisions is considered as independent variables affecting design results and, therefore, product quality (Fig. 1, upper left box).

It seemed that the collection of all distinct design decisions and their classification in a certain framework that could even be project-dependent, might be an adequate approach. The result was a hierarchical classification model for the system design development process [Ebert and Riegg, 1991]. For improved understanding of this hierarchy we put its different levels of abstraction on top of the conventional design steps (Fig. 2). These design steps symbolize the process of selecting distinct design decisions out of the complete classified set of all possible aspects. The original requirements are refined during the system design into external (nonfunctional) constraints and goals on the one hand and solution-inherent (functional) aspects on the other. The software requirements analysis separates into system environment or design description, and system structure, internal structure, system communication, or dynamic structure, respectively. Finally the preliminary design produces system behavioral issues that

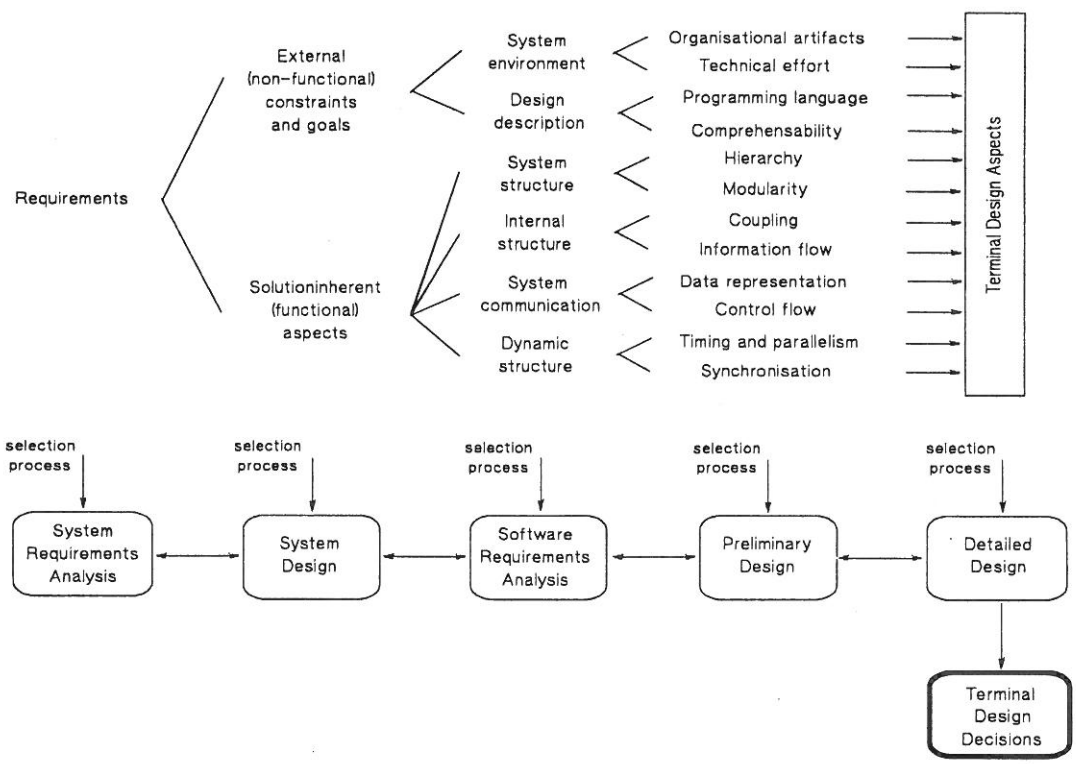

Fig. 2. Hierarchical model for the classification of possible design decisions

we divided into 12 classes [Ebert and Riegg, 1991]:

1. Organizational Artifacts (type and degree of organizational and personal constraints influencing the development);

2. Technical Items (system environment and technical constraints).

3. Programming Language (qualities, particularities, or peculiarities of the language to be chosen);

4. Comprehensibility (representation and description of the system);

5. Hierarchy (structure in which the specification objects are ranked into levels of subordination);

6. Modularity (degree to which the system is composed of discrete objects);

7. Coupling (manner and degree of interconnections between software modules);

8. Information Flow (use of data objects and their flow);

9. Data Representation (representation of data objects and the individual allocation);

10. Control Flow (the intended sequence in which operations or actions are performed);

11. Timing and Parallelism (dynamic issues of the system, e.g. time constraints, parallel or distributed structures, real time issues);

12. Synchronization (influence on and control of system dynamic issues).

All decisions that emerge out of system design and preliminary design influence the detailed design and vice versa. The 12 classes are subsequently refined into 35 subclasses (e.g. the hierarchy class into "system hierarchy", "refinement hierarchy", "calling hierarchy") and these subclasses are further divided into 140 groups (e.g. the system hierarchy subclass into "extension", "ree structure", and "homogeneity"). The terminal design decisions are on the last level of this classification tree. We investigated around 300 different terminal design decisions which are basically independent of the specific software development environment [Ebert and Riegg, 1991]. Application to a specific CASE environment (EPOS; *Engineering and Project-Management Oriented Support System* [EPOS, 1991]) is described in ch. 3. The tailoring process of selecting appropriate and reduced classification models that finally lead to individual goal-dependent sets of metrics is illustrated in Fig. 3. Transformation of all specified design decisions to the EPOS environment showed that some environment independent decisions simply could not be projected into this environment because of distinct limits (e.g. no support by the specific design specification language). Other decisions resulted in multiple but similar entries after the transformation process that always looses some information, thus being mostly canceled. The effect of this process was a reduction of the 300 original decisions to slightly more than 100 design decisions useful and applicable to the EPOS environment.

All these terminal design decisions are measurable if intended, hence providing a vast set of potentially quantifiable design decisions! Since we are focusing on design decisions that are measurable early in the development process, we won't give any proposals for measuring process related aspects (resources, software testing) in this article. The advantage of a tool independent classification is that the design decisions can be applied to any environment and later on applied to the specific tools. This is highly necessary in heterogeneous CASE environments where different modular tools are combined for a distinct project. In addition, the investigation of design decisions can be (and actually is) used for all different design methodologies.

A lot of single design decisions are derived during the complete design process: Time constraints of processes, data coupling between different design objects, control of atomic tasks, interaction of hardware components (e.g. integrated circuits, sensors, etc.), or the programming language and its specific qualities and support tools are just a few examples for these complex interdependencies. The hierarchical classification implies that certain design decisions might influence different areas. The data flow and its impacts, for example, affect coupling, but also information flow, control flow, and even modularity. However, the priorities are distinct: in one case the length of the data paths throughout the modules and procedures are of interest, while in another case the data objects and their components is of interest. Some

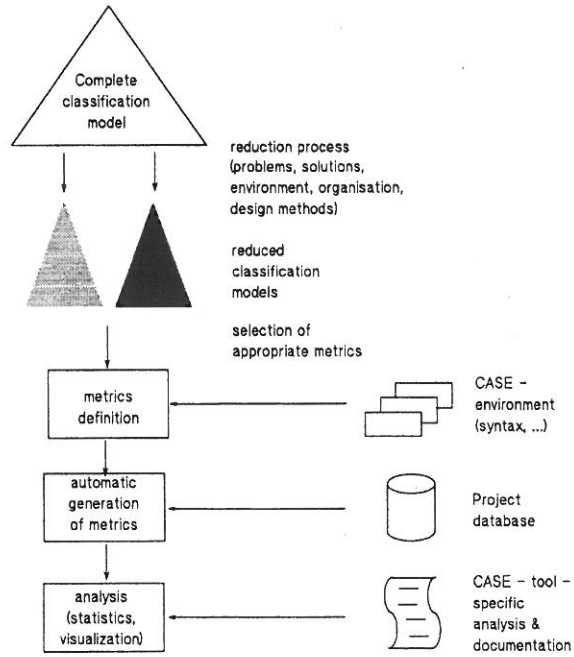

Fig. 3. The reduction process of tailoring a specific classification framework

of the refined design decisions are highly environment dependent and deeply influenced by the project under development. For example, mechanisms for deadlock detection and prevention are important for the design of embedded real-time systems, while they are of no interest in an almost linear scientific program without any parallel resource allocations.

Complexity factors are determined by several different metrics that we combined to a single vector. Thus, the multifactorial appearance of complexity is considered. Again, it needs to be mentioned that this measurement vector is selected according to the domain area of problems to be analyzed, namely in industrial automation with lots of interfaces and real-time aspects. Another approach in defining a measurement vector for the size of data-centered business systems is proposed in Evanco and Lacovara, 1994. Discussions on recent CASE workshops about the choice of metrics CASE vendors should include in their products showed disagreement among users about what metrics they precisely wanted. In a preliminary analysis our measurement vector consisted of thirty elements that were further reduced to sixteen single elements according to the following criteria:

- Applicability to the techniques used in our approach of developing real-time automation systems. All complexity metrics need to be applicable to design specification languages, both textual or graphical, and to structured real-time languages (*Ada*, *PEARL*).

- Traceability of distinct complexity factors independent of the method and technique that is applied during a development step. The candidate metrics should measure one distinct factor for different design languages.

- Statistical evaluation by means of parametric statistical techniques should be permitted. The metrics hence should have a distinct distribution and need to be at least on a ratio scale.

- Common properties for metrics should be fulfilled. These properties include robustness towards the underlying complexity factor, specificity of the contributing complexity factor, reproducibility which results in automatic measurement, comparability of different metrics, or usefulness of their application in real projects.

To reduce the factor dimension of the complexity measurement vector we applied concepts of multivariate statistics (factor analysis and cluster analysis) [Ebert, 1992]. They are primarily used in the analysis of complicated sets of data and a priori unknown relations. Such techniques for analyzing complicated sets of data

may be regarded as defining a set of new variables derived from the original variables. This reduced set of variables is intended to focus on certain aspects of the original data without losing too much information from it [Munson and Khoshgoftaar, 1990].

The resulting metric vector consists of the following sixteen elements (affiliated complexity factors based on Fig. 2 are in italics and measurement description; both in parentheses):

1. system depth (*hierarchy*; supremum of number of refined hierarchy levels of modules).

2. module size (*modularity*; number of functions per modular unit).

3. functionality (*modularity*; number of coupling references among functional units divided by all functional units in a module).

4. network size (*coupling*; number of all coupling references including data, time, functional coupling etc.).

5. functional coupling (*coupling*; number of coupling references among functional units).

6. global data flow (*information flow*; number of all globally used data objects).

7. external interfaces (*information flow*; number of input / output elements).

8. operand count (*data representation*; number of unique data elements).

9. cyclomatic complexity (*control flow*; number of binary decisions in the control flow).

10. nesting degree (*control flow*; average number of nesting levels in complete control flow).

11. nesting depth (*control flow*; supremum of nesting depth).

12. parallelism degree (*timing and parallelism*; number of potentially parallel tasks).

13. synchronization degree (*timing and parallelism*; number of synchronization elements).

14. overall volume (*volume*; number of textual lines and / or independent, distinctive graphical elements).

15. statement density (*comprehensibility*; size of functional volume divided by size of non-functional, descriptive volume).

16. denotational length (*comprehensibility*; average length of object names).

The rules for accounting the complexity measurement vector are kept stable since an initial phase when they have been selected from a broader collection of metrics. This is important in order to achieve reliable and reproducible metrics. Communication problems caused by imprecise and changing definitions of metrics may cause an entire measurement program to fail. For example, if overall volume is reported in lines of code, this metric is dependent on whether comments or blank lines are counted. The difference in the figures could be in the order of two or three to one [Fenton, 1991].

As we were also interested in validating this selection of complexity metrics, we classified them according to the fulfillment of nine criteria according to Weyuker, as discussed in [Zweben, 1990]:

1. Nonidentity: Not all software should be rated equally.

2. Finiteness: The metric must give the same value to finitely many programs.

3. Noninjective: At least two different pieces of software are rated as equally complex.

4. Antiextensionality: Two pieces of software with same functionality are assigned different complexity values.

5. Monotonicity: The software consisting of several components is at least as complex as one of its components $((\forall P)(\forall Q)(|P| \leq |P;Q| \wedge |Q| \leq |P;Q|))$.

6. Anticomposition: Adding the same software to two components with equal complexity might result in two pieces of software of different complexity $((\exists P)(\exists Q)(\exists R)(|P| = |Q| \wedge ((|P;R| \neq |Q;R|) \wedge (|R;P| \neq |R;Q|))))$.

7. Antipermutability: Changing the order of contents in two components might result in different complexity.

8. Renaming: Components with identical contents and different naming (e.g. variable names, etc.) have the same complexity $((\exists P)(\exists Q)((P \equiv Q) \wedge (|P| = |Q|)))$.

9. Integration: After combining two components the resulting complexity might exceed the sum of complexities of the components.

A formal description of some properties of each complexity vector component is given in table

complexity metric	**properties of complexity metrics** according to Weyuker									**range of** values
	1	2	3	4	5	6	7	8	9	
system depth	•		•	•	•	•		•		$\in \mathbf{N}_0$
module size	•	•	•	•	•	•		•	•	$\in \Re_{0+}$
functionality	•	•	•	•	•			•	•	$\in \Re_{0+}$
network size	•		•	•	•	•		•		$\in \mathbf{N}_0$
functional coupling	•		•	•	•	•		•	•	$\in \mathbf{N}_0$
global data flow	•		•	•		•	•	•		$\in \mathbf{N}_0$
external interfaces	•	•	•	•	•			•		$\in \mathbf{N}_0$
operand count	•	•	•	•	•	•		•		$\in \mathbf{N}_0$
cyclomatic complexity	•		•	•	•			•		$\in \mathbf{N}_0$
nesting degree	•		•	•		•	•	•		$\in \Re_{0+}$
nesting depth	•		•	•	•	•	•	•		$\in \mathbf{N}_0$
parallelism degree	•		•	•	•	•		•		$\in \mathbf{N}_0$
synchronization degree	•		•	•	•	•		•		$\in \mathbf{N}_0$
overali volume	•	•	•	•	•			•		$\in \mathbf{N}_0$
statement density	•		•	•		•				$\in \Re_{0+}$
denotational length	•		•	•		•				$\in \mathbf{N}_0$

description: • criteria fulfilled

Tab. 1. The Complexity Vector and Some Properties of its Components

1. The first column names the 16 components of the vector. Weyuker's nine properties are assigned to each component in the following columns. The range of possible values (in terms of being either integer or real) is presented in the last column.

All suggested metrics are on a ratio scale level. Statistical methods to investigate relationships among data sets are defined for different scales and can be correctly applied up to the scale level they are defined for [Fenton, 1991; Zuse, 1991]. For example, parametric tests that require the calculation of distances (means, standard deviations, correlation coefficients) are defined for the data given at least in one interval level of their measurement scale. It is possible to lower the level of the scale, which results in an information loss, however it is prohibited to scale up. Parametric multivariate statistical techniques usually require a ratio scale. s we wanted a scale level allowing such parametric statistical techniques, all metrics are on a ratio scale level. For the purpose of comparability all metrics had been defined in the way to include zero as the lowest value of complexity, which seems a reasonable property from a measurement theoretic point of view [Fenton, 1991; Zuse, 1991].

Not all of Weyuker's properties are fulfilled by the metrics, because for example, they investigate exactly what Weyuker excluded (e.g. criterium no. 8 is not fulfilled by statement density and denotational length that measure psychological or comprehensive complexity). Failure in fulfilling the first five criteria is more critical and in suchcases metrics should be subjected to further analysis. Cyclomatic complexity or global data flow are such candidate metrics. Criterium no. 2 is not fulfilled by such metrics that are too coarse, while criterium no. 5 is even more important due to the — intuitive — requirement, that complexity can not be reduced by adding complexity. In fact, our metric vector only violates these criteria in cases where metrics are normalized or where comprehensibility is measured.

We realized that, although such criteria are described as necessary — and not sufficient — for "acceptable" metrics, they still need to be improved. Besides considering structural complexity such criteria should also treat aspects of pure psychological complexity. For example, the property suggesting that functional identical programs should result in identical metrics needs to be expanded towards considering descriptive parts of the programs under evaluation.

Functional identity simply does not deal with naming conventions or length of remarks. Especially the comprehensibility metrics treat this suggestion (e.g. by classifying those programs or designs as more complex, having fewer descriptive parts, or that have non-spelling object names), thus not fulfilling this original property.

Criteria for validating software metrics need much more attention than just discussing the few articles available, which deal with pure structural complexity, over and over again. Validating metrics, according to such formal criteria, seems reasonable from a theoretic point of view, while validation based on what is intended to measure (i.e. based on quality goals) is of practical importance. Effective validation requires appropriate use of methods such as inspection, reviews, testing, or statistical comparisons based on quality models and quality goals that provide a baseline for design and hence by their outcome show the applicability of design metrics as quality indicators.

3. The Integration of Design Metrics in a CASE Environment

To obtain insight and real benefits of metrics, it is necessary to build a tool that helps collecting metrics automatically and then investigate real-world projects. We implemented metrics for different groups of the given classification model (e.g. for information flow, control flow, hierarchy, modularity, comprehensibility, etc.) in a commercially available CASE environment. The metrics were selected according to mutual independence and usefulness. They represent the described sixteen element measurement vector. In reporting measurement data from diverse development projects, we included the raw data (e.g. complexity metrics for different products of the life-cycle phases under investigation) and additional qualitative results (e.g. number of faults and changes both due to incorrect or missing realization of requirements and effort for each product to be developed). It is important to report such qualitative factors in order to improve interpretation of metrics.

The 16 metrics of the measurement vector were selected according to mutual independence and usefulness. They were integrated into EPOS because this state of the art CASE environment supports all development phases of a project (from requirements formulation and analysis through system development, coding and testing, to system implementation and maintenance including project management support for all phases) for the entire software/hardware system. Since this environment permits the use of various design methods (e.g. event-, function-, module-, data-flow-, data-structure-, or even device-oriented procedures), it is possible to trace the influences of different approaches or the results with respect to design metrics.

The EPOS tool [EPOS, 1991] represents a CASE environment, which is used in over 1200 industrial installations all over the world, mainly in Europe but also in the US. It is used in all kinds of software projects and covers all phases of the software development process. The EPOS environment provides three different specification languages:

- A requirements specification language (called EPOS-R).

- A system design specification language (called EPOS-S) to represent the design at different levels of abstraction. This language includes seven design objects to hierarchically model tasks or procedures, data items, conditions, events, interfaces and hardware components (Fig. 4). This is the EPOS language relevant for software design and therefore design metrics.

- A specification language (called EPOS-P) to describe project management information.

System design with the specification language EPOS-S is highlighted in Fig. 4 which shows an excerpt of the EPOS-S design model. According to a top-down approach, all design information is transformed into the combination of seven distinct design objects. The boxes in Fig. 4 represent these design objects, e.g. modules, actions (for tasks, functions or procedures), data objects, conditions, events, etc. Out of several existing design methodologies supported by EPOS, this example presents a function- and module-oriented approach. The upper levels of the design model are defined during preliminary design, the lower levels are defined during detailed design. The definition of specific design objects takes place by invoking these objects, placing references between them, and by qualifying these objects with additional attributes

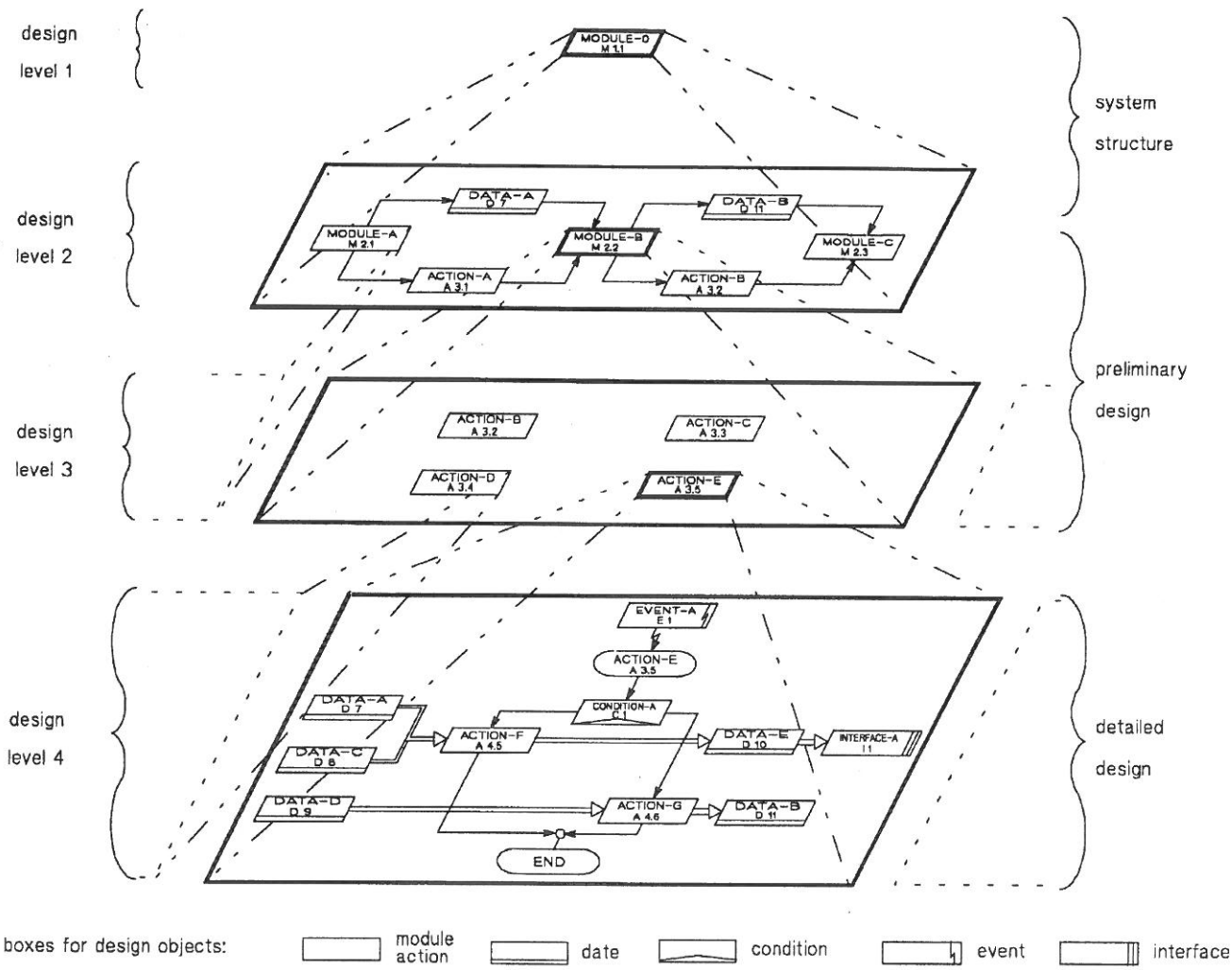

Fig. 4. Hierarchical design model for a top-down approach used by EPOS-S

(e.g. timing constraints, synchronization, control flow).

After entering the specified design information into the EPOS data base, it can be retrieved and evaluated by a number of tool systems to support the designers. The design measurement tool system has been implemented on the same basis as all other analysis tools. Therefore it can be used in a highly interactive fashion during the complete design process. Problem areas being discovered while designing the top levels can easily be changed before they might cause any damage on the lower levels. Application of the tool to industrial projects supported the idea of classifying design decisions in a hierarchical model. Thorough analysis of relationships among measured design decisions during the preliminary design showed influence on the characteristics of lower levels.

Concerning the technical integration of our measurement tool system into the EPOS environment, we distinguish two interfaces (Fig. 5). The first is given the retrieval functions of the metrics to get the design information and the second is given by output functions of the metrics. With the help of these interfaces we are able to fulfill the following requirements:

• *Use of a given retrieval layer as the input interface.* All other components of the CASE system use the same retrieval layer, e.g. the analysis tool system for the analysis of completeness, type conflicts and isolated objects. Thus, less maintenance effort is necessary for the measurement tool system in the data retrieval part. If there are any changes in the database it is only the retrieval layer that has to be adjusted but not the

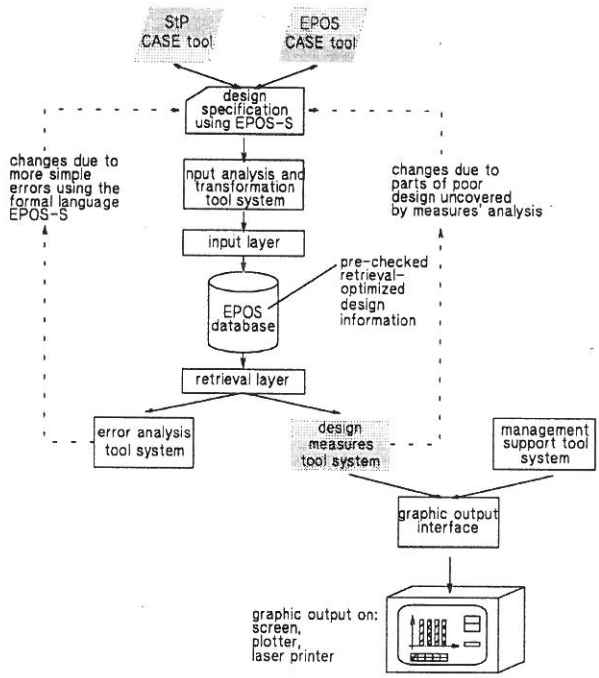

Fig. 5. The EPOS environment and the integration of the design metrics tool system

complete tool system.

• *Use of the pre-analyzed and retrieval-optimized information of the EPOS database.* It is therefore possible to develop simpler measurement programs because a lot of normally necessary (syntax-) analyses can be omitted. In addition, system performance is improved.

• *Use of a given output layer to generate the graphical display.* Some other analysis components in EPOS apply the same interface, e.g. the project management support system for the documentation of cost and capacity analyses. Changes in the hardware environment can hence easily be managed. If some additional devices are to be used by all output functions, only the output layer has to be changed.

Besides, the complete measurement tool system is implemented in the same source code language (*Syslan*) as all other functions of the EPOS environment. The effort for developing this measurement tool and for integrating it into EPOS took around three person years. This includes statistical and visual data deployment.

Based on the underlying idea of classifying design decisions, selecting appropriate design metrics, and implementing them according to a modular approach — including the use of already existing database retrieval mechanisms or graphical output systems — , it is feasible to transfer such a tool to other environments. If the whole EPOS environment including the measurement tool system needs to be ported to another platform, only the compiler for that source code language has to be adapted to the new processor and the new operating system. Then the EPOS environment is to be compiled and linked, and it will run in any other environment.

The open architecture approach of the design classification and measurement environment permits the combination with other CASE tools. Using the measurement tool in another environment only requires a translator to convert design representations into the standard form expected by the EPOS database (e.g. with *lex* and *yacc*). Then the measurement tool can be applied to standard representation to compute various metrics. The EPOS-S design language is used as the target language for such transformations because of its variety of constructs supporting different design methodologies and real-time system design (including software and hardware components and their interactions). Currently a translator is available for the *Software through Pictures*™ CASE tool.

After or while entering design information in

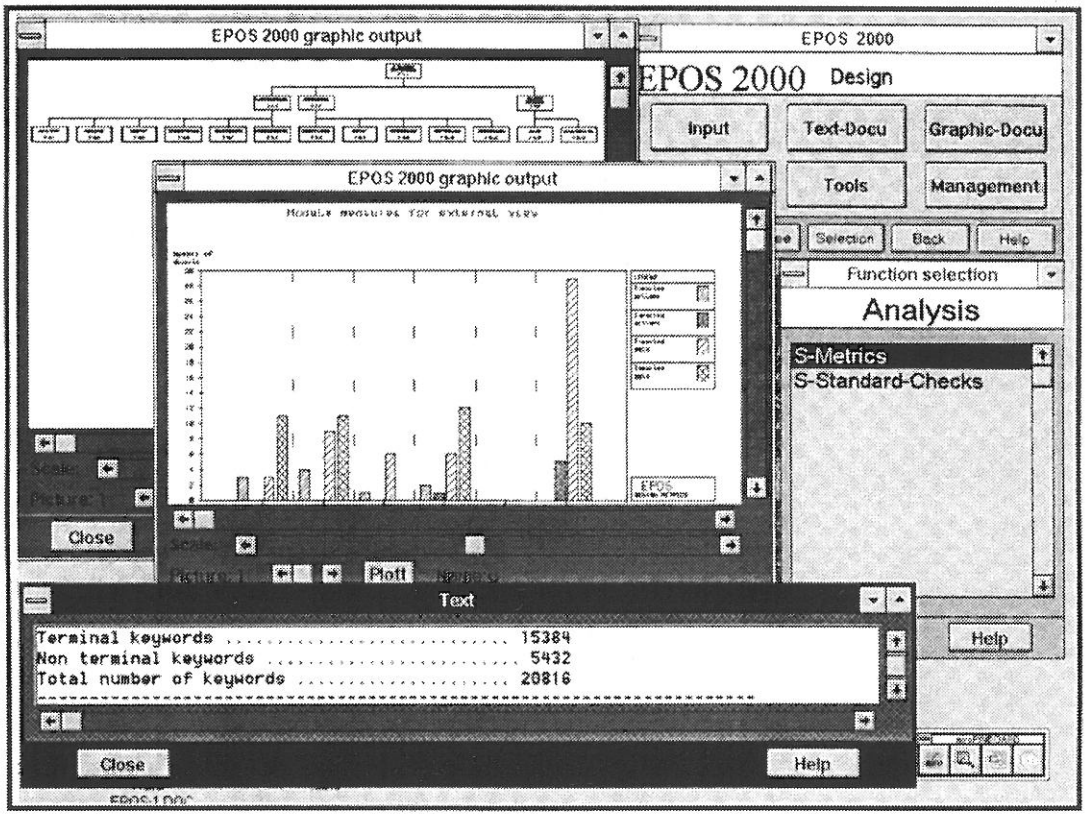

Fig. 6. Typical screen of the metrics environment within EPOS

the project database, the user can activate the measurement tool system by using some simple menus. The results are obtained almost immediately (of course, the reaction time dependens on the project and database size). A typical screen while working with the design metrics tool is presented in Fig. 6. For this application, we chose a PC setting under MS windows, however due to the chosen graphical input/output layer, the design metrics looked similar under X-Windows on a workstation.

4. Experience With the Tool– Exemplary Results

For better understanding, the results of applications in several industrial real-time projects are briefly described. One application included around 50 modules with altogether 100,000 LOC (*lines of executable code*) to be explained later on. First of all, we investigated the distribution of raw data with the Kolmogoroff-Smirnov and the chi-square tests. Despite of (ordinary level scaled) defectiveness and maintainability, normal distribution can be assumed for all described complexity metrics with $\alpha = 0.01$ which is sufficient for exploring parametric statistical analyses. Design metrics correlated with each other according to different underlying complexity factors (which in turn provides some validation for the factorial design complexity model). For example, total volume, system depth, functionality, total coupling, functional coupling, and unique operands correlated with values higher than 0.8 (confidence level: $\alpha < 0.01$). Other complexity factors did not correlate at all (e.g. parallelism, synchronization, depth of nesting, or denotational length). Statement density was even negative correlated with most complexity metrics, thus showing that complex designs need an overhead for documentation. Effort (in terms of days spent on design) correlated high with most complexity metrics. An example is given in Fig. 7 which groups effort (vertical axis) over total volume (horizontal axis) with additional classification for functionality or total coupling, respectively.

The metrics proved a high dependence on the design method actually applied, hence providing guidelines for a latter application of source

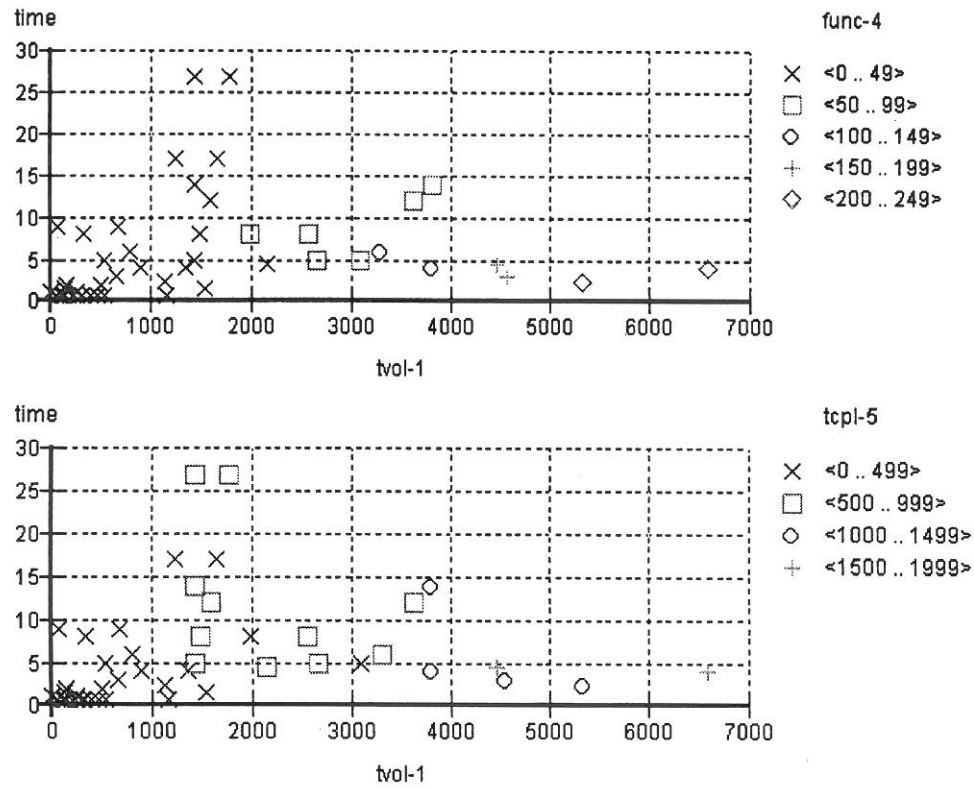

Fig. 7. Design metrics from real-time projects: effort spent on each module versus total volume with indication of functionality (top) and total coupling (below)

code metrics. Again, Fig. 7 shows that those designs with many small procedures (or 'modules' in a Parnas-type definition) take less effort than the big modules that contain too many functions without further reduction. Design metrics also indicated outlying or critical design objects (e.g. some procedures and tasks with an extreme volume (more than 300 lines of PDL like code without counting comments), too many relations to other design objects (over 100 references), and high cyclomatic complexity. Another design had an extremely broad and flat hierarchy (less than 10 modules used around 200 procedures and tasks that were grouped in just 4 design levels), hence indicating complicated calling and refinement relationships.

After having compared several designs of real-time systems [Ebert, 1994] we found out the following relations between design complexity and product quality:

- High cyclomatic complexity shows an exploding number of combinations of control paths and a large number of competing conditions in a module. Both result in reduced testability (e.g. C1 coverage cannot be achieved) and maintainability which of course increases the risk of not detecting faults in the software. For the same reasons nesting degree and nesting depth should be reduced to a level that permits testing and comprehension.

- High global data flow, coupling or operand count show a processing overhead from a large volume of data. Testability is reduced because test environments are difficult to produce and populate, test productivity is reduced for the same reasons and maintainability is reduced because changes might cause effects in many other modules.

Based on our investigations we extracted several design suggestions in order to achieve a low complexity design:

- Apply modern design and programming practices during the design and do not wait "till coding". This includes such principles as top down requirements analysis and design, structured design with modules, hierarchies, and location of data items, functions, and events for synchronizing parallel tasks. Structured design with small modules result in reduced control flow nesting and less decisions.

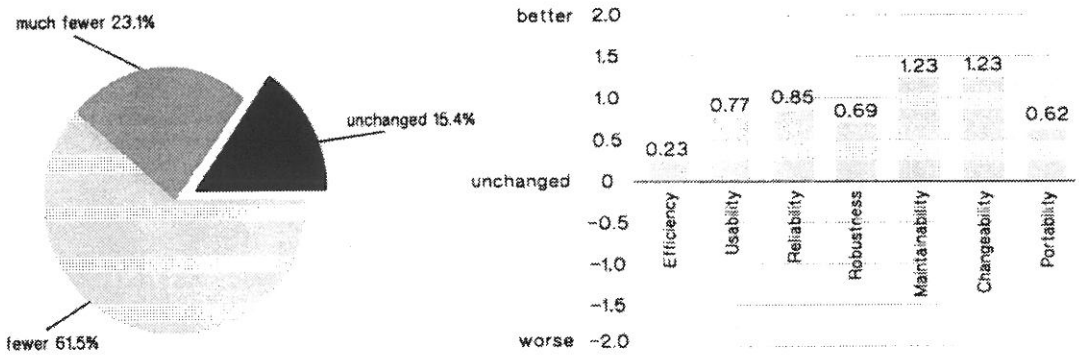

Fig. 8. Results from a survey: percentage of system failures due to design or specification faults after using the EPOS CASE tool (left side); the improvement of overall system quality according to different criteria using the EPOS CASE environment (right side)

• Minimize the module connections by reducing their fanout to around three to five. Of course design goals such as small modules and low coupling are influencing each other and thus require clear priorities on design goals.

• Reduce the control flow complexity by limiting the nesting depth to less than four and limiting the number of control flow decisions or the cyclomatic complexity to ten for single procedures or tasks.

• Try to achieve design objects' refinement and calling hierarchies that resemble a S-shape curve when the cumulative number of sub-objects is printed against the number of the hierarchical level. At the lower end of this curve towards the top objects, its growth should change smoothly.

• Sustain a high comprehensibility level by providing a sufficient length of descriptive parts in all design objects and object names with meanings rather than enumeration, such as "module-1". The descriptions should include functionality, related objects, used data items, date, author, test cases to be performed, requirements fulfilled, management activities and staff connected with this object, and so on.

• Use modern design tools, especially CASE environments, and apply design methods that fit your problems [Rugaber et al, 1990]. This seems to be clear for programming languages but not for design methods. For example, real-time automation projects need other design techniques and programming languages than database management systems. There are CASE tools around for method support. If they don't fit to your specific approach (e.g. method support) use meta-CASE tools. Once you have derived a detailed design, translate it automatically to the appropriate programming language. Thus, you are ensured that no information is lost, the descriptive parts are the same, and that there is no unnecessary workload to be performed.

The measured values were analyzed with respect to boundaries (minimum, maximum), intervals, deviations from the average, and correlations between them. The interpretation was performed according to these criteria and used as follows:

• During preliminary design, the metrics and their statistical evaluation (regarding similar projects) are taken to distinguish different designs (e.g. alternative approaches, division into subcomponents).

• During reviews at the end of each phase, the metrics are taken as indicators for weak components (e.g. inadequate calling hierarchy, unsatisfying system description) and as indicators for process management (timely ordered hierarchy or volume metrics).

• After applying such metrics to different, however similar projects, the obtained statistical results can be used to define intervals or limits for metrics, in order to increase the quality.

A recent survey of 25 major projects (effort: 2 . . . 48 pers. years) at 23 large European com-

panies that were developed with the EPOS environment reported improvements in quality (Fig. 8; results not yet published in English). Most important was the high ranking of early fault detection and removal during system and preliminary design, which contributed indirectly to an increasing overall project productivity. This survey also proved that system quality assurance is actually provided during the design in the early phases of the life cycle instead of late evaluative techniques, thus indicating the importance of design quality assurance and design quality control. Problems combined with the measurement tool system that have been observed up to now were usually related to the lack of knowledge about design practices and improvements. We have seen projects that are still integrated in an unstructered Cobol style just because the project manager has always done it this way. The result of this observation is the development of courses for software quality assurance with complexity metrics.

5. Conclusion

Complexity is a not-so-warm feeling
in the tummy.
Bill Curtis (1979)

Most complexity metrics have been designed without regard to the problem domain and the programming environment. There are many forms of complexity and there are a lot of design decisions influencing the complexity of the product. More important, metrics must be applied early in the development life cycle to provide trade-off. Once code is written or a detailed design of a system has been achieved, much effort and time has been invested into what might be an architectural unsound system. It is hence at the top level design stages, when functionality in the system is being partitioned and the component interfaces are being defined, that software quality prediction is most useful to the designer.

In order to obtain real benefits of design metrics, we developed a hierarchical classification model of different groups of design decisions in 12 classes that cover over 300 single terminal design decisions. With the model it is fairly easy to extract a reduced model of design decisions that covers all design decisions of a specific project. These measurable design decisions are fundamentals for the application of design metrics. The advantage of this approach is the simplicity of selecting project-oriented design metrics out of a set of given design decisions for all fields related to product complexity. Since design involves making choices among alternatives and too often the underlying rationale for such decisions is lost, the model also allows to trace design decisions through the development process.

This paper presents an approach to integrate software metrics techniques into a commercially available CASE environment. The proposed design decisions model is a useful way to avoid the lacks of pure statistical analysis and interpretation of software metrics and to decide which metrics to implement. Since today's software engineering is dealing with the system's software and hardware development, a CASE tool supporting both interacting components was selected for the integration of the measurement tool. The implementation in the EPOS environment showed typical ways of coping with already existing databases and interfaces for the complete system life cycle. Our method to integrate a measurement tool system into CASE environments illustrates a way to minimize the efforts for implementation and maintenance of such a tool system and show how to be able to deal with changes in future requirements for such tools and their individual interfaces. By transforming the design information into a distinct target language as mentioned in this paper, it is possible to integrate such measurement techniques into other environments as well.

With an early analysis of software products we are able to give system engineers helpful hints to improve their designs early in the life cycle. Advantages using the proposed design metrics tool together with the supporting CASE environment include:

- reduction of costs by predicting critical areas in a particular design;
- early detection of abnormalities based on average during design and by showing complexity trends as early as top level design;
- ability to compare different designs and thus provide decision support during design.

By following the given suggestions we could improve designs and achieve a better design and product quality in terms of such quality items as understandability, reliability and maintainability. Of course, much more research is necessary in order to provide complete guidelines for achieving high quality designs. The basic step, however, still is the measurement and evaluation of software complexity as early as possible: during the software design phase when the most expensive faults are induced. By making software engineers aware that there are suitable techniques and tools for analyzing their designs this could be yet another small step to overcome the omnipresent software crisis.

Acknowledgments

I am indebted to Andreas Riegg for his preliminary work in developing fundamental parts of the tool. Tool integration and support for realizing the design metrics environment within EPOS has been provided by GPP, Munich. The DFG (German National Science Foundation) is gratefully acknowledged for the financial support of this research project.

References

AL–JANABI, A. and E. ASPINWALL, (1993): An Evaluation of Software Design Using the DEMETER *Tool. Software Eng. Journal*, Vol. 8, No. 6, pp. 319–324.

CARD, D. L. and R. L. GLASS, (1990): *Measuring Software Design Quality*. Prentice Hall, Englewood Cliffs. NJ, USA.

EBERT, C., (1992): Visualization Techniques for Analyzing and Evaluating Software Measures. *IEEE Trans. Software Engineering*, Vol. 18, No. 11, pp. 1029–1034.

EBERT, C., (1994): Assessing the Impact of Structured Real-Time Design Methods. *Journ. of Microcomputer Applications*, Vol. 17, pp. 287–297.

EBERT, C. and A. RIEGG, (1991): A Framework for Selecting System Design Metrics. Proc. of the Int. Symp. on Software Reliability Engineering. IEEE Comp. Soc. Press, Los Alamitos, CA, USA, pp. 12–19.

EPOS OVERVIEW, (1991). Distributed by: SPS Software Products and Services, Inc., 14 E. 38th St., New York, NY 10016, USA.

EVANCO, W. M. and R. LACOVARA, (1994): A Model-Based Framework for the Integration of Software Metrics. *Journal Systems and Software*, Vol. 26, No. 1, pp. 77–86.

FENTON, N. E., (1991): *Software Metrics: A Rigorous Approach*. Chapman & Hall, London, U.K.

GUINDON, R., (1990): Designing the Design Process: Exploiting Opportunistic Thoughts. *Human-Computer Interaction*, Vol. 5, pp. 302–344.

MELTON, A., D. A. GUSTAFSON, J. B. BIEMAN and A. L. BAKER, (1990): A Mathematical Perspective for Software Measures Research. *Software Engineering Journal*, Vol. 5, No. 5, pp. 246–254.

MUNSON, J. C. and T. M. KHOSHGOFTAAR, (1990): Regression Modelling of Software Quality: Empirical Investigation. *Information and Software Technology*, Vol. 32, No. 2, pp. 106–114.

PORTER, A. A. and R. W. SELBY, (1990): Empirically Guided Software Development Using Metric-Based Classification Trees. *IEEE Software*. Vol. 7, No. 3, pp. 46–54.

RUGABER, S., S. B. ORNBURN and R. J. LEBLANC, JR., (1990): Recognizing Design Decisions in Programs. *IEEE Software*, Vol. 7, No. 1, pp. 46–54.

SELBY, R. W. and V. R. BASILI, (1991): Analyzing Error-Prone System Structure. *IEEE Transactions on Software Engineering*, Vol. 17, No. 2, pp. 141–152.

STARK, G., R. C. DURST and C. W. VOWELL, (1994): Using Metrics in Management Decision Making. *IEEE Computer*, Vol. 27, No. 9, pp. 42–48.

ZUSE, H., (1991): *Software Complexity: Measures and Methods*. De Gruyter. Berlin.

ZWEBEN, S. H., (1990): Evaluating the Quality of Software Quality Indicators.*Proc. 22nd Symp. on the Interface, Statistics of Many Parameters*. pp. 266–275, Springer, Berlin.

Received: February, 1995
Accepted: July, 1995

Contact address:
Christof Ebert
Alcatel SEL AG
Communication Systems
Lorenz str. 10
D–70435 Stuttgart
phone: +49–711–821–43955
e-mail: cebert@stgl.sel.alcatel.de

CHRISTOF EBERT holds a Ph.D. with honors from the University of Stuttgart. He is with the quality strategies department of Alcatel SEL in Stuttgart. He has published over thirty refereed papers in the area of software metrics, real-time software development and CASE support for such activities. His current research topics include software process analysis and improvement, software resuability and productivity analysis.