

# Evaluating Evolutionary Software Systems

Teade Punter, Adam Trendowicz, Peter Kaiser

Fraunhofer IESE, Sauerwiesen 6, D-67661 Kaiserslautern, Germany  
{ punter, trend, kaiser } @iese.fhg.de

**Abstract.** non-functional requirements (NFRs) of software-intensive systems that are under continuous evolution should be evaluated during early development phases in order to be able to improve those systems and achieve ‘time-to-market’. However, current evaluations are often done during late stages, like coding and testing. In this paper we propose an approach to evaluate NFRs earlier. The requirements for this approach are the use of flexible and reusable quality models, which can deal with little data, that are transparent and measurement-based. Our approach, called Prometheus, is a way of modeling NFRs that should cope with those requirements. Prometheus applies the quality modeling concept from the SQUID approach, the probability concept of Bayesian Belief Nets (BBNs) and the specification concepts of the Goal Question Metric (GQM) approach.

## 1. Introduction

With the complexity of the software that is developed nowadays, also the attention for non-functional requirements has increased. Non-functional requirements concern the reliability, maintainability, efficiency or portability of a software-intensive system. Especially in complex systems where software is embedded, like automotive systems or medical devices, non-functional requirements have great influence on user-performance. For instance, rebooting a control panel in a car that takes one minute is unacceptable.

Especially in market-driven software development, where embedded software is developed the necessity of paying attention to non-functional requirements is already known. Market-driven software development focuses on markets instead of particular customer or (group of) user(s). An example is the TV-market. Dealing with software of TV-tuners requires considerable attention for Reliability issues. Putting the software once in a TV does not allow future improvement of the software after it has been sold. Of course, ‘maintenance on the fly’ is a possibility [1], but economical approaches for this, that have wide user-acceptance too, have not been proved yet.

A typical feature of these kind of software-intensive systems is their evolution context. *Evolution* is hereby considered as the evolution of requirements, systems and system families, system architectures, individual components, resource constraints (concerning timing & memory requirements) and underlying hardware. The existence

of evolution in software systems is already known from the 1970's [2]. The increasing pressure of time to market, combined with the production of complex software in an evolution context pushes industry in estimating non-functional requirements as early as possible in the development process. Correcting or improving the software in an early phase is better than in front of the acceptance test. Instead evaluation of software intensive systems during their whole life cycle is necessary. This is derived from the requirements applied when generally designing reliable products in cost and time driven market, namely [3]:

- risks and potential problems are identified and resolved as much as possible in the early phases of the development process;
- actual capabilities of products, in terms of its functionality and non-functional quality are validated as soon as possible in the development process;
- where a mismatch between prediction and actual capability exist this mismatch is investigated on root-cause level and this information is deployed to the relevant actors.

This paper deals about the evaluation of NFRs in software-intensive system by focusing on the possibilities for early predicting system's quality. Section 2 identifies the problems in evaluating NFRs in software-intensive systems. We will argue that specific requirements for quality models should be set. These requirements are presented in section 3. The sections 4 and 5 define the Prometheus approach that should fulfill these requirements.

The presented research is a reflection of the work that the authors conduct for the ITEA Empress project<sup>1</sup>. The project aims at developing a methodology and process for real-time embedded software development that supports management of evolution in a flexible and dynamic way. The project has started in January 2002 and will run till December 2003.

## 2. Evaluating Non-Functional Requirements

This section identifies the problems in evaluating NFRs in software-intensive systems. This is done by paying attention to the “why”, “what”, “how” and “when” questions regarding evaluating NFRs.

Why – it has already been stated in previous section that the increasing pressure of time to market, combined with the production of complex software in an evolution context pushes industry in estimating non-functional requirements as early as possible in the development process. Evaluation NFRs is required to control the development process as well as the complete systems' lifecycle to produce real time and embedded software systems that meet the customer's expectations.

*What is evaluation of NFRs?* – it is the systematic examination of to which extent the non-functional requirements are fulfilled by (parts of) the software intensive system.

---

<sup>1</sup> Empress stands for: Evolution Management and Process for Real-time Embdeded Software System; <http://www.empress-itea.org/index.html>

Evaluation NFRs has to do with verifying requirements [4], which is the examination, analysis, test, or demonstration that proves whether a requirement has been satisfied, however it is also related to validating the requirements to ensure that the set of requirements will be consistent and that a real-world solution can be built that satisfies the requirements, as well as that it can be proven that such a system satisfies its requirements.

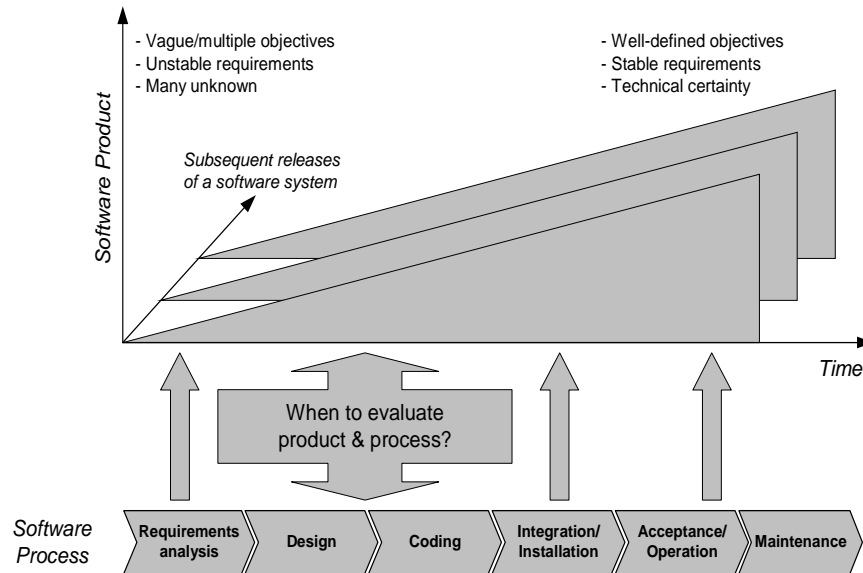
*How is evaluation conducted?* – code metrics assessment is a major approach to evaluate NFRs. It is the Static analysis tools like QAC/++ or Logiscope are used for this code measurement. These tools are able to identify files of source code that are not conformant to the specified quality characteristics. Attributes that are often dealt in code metric assessments are: cohesion, and maintainability, fault proneness [5]. The tools can also be used to baseline the current state of the software.

*When to evaluate?* – code metric assessment is one of the evaluation approaches. Another approach is architecture assessment that is applied during the design phase when architecture is constructed. Major approaches for architecture assessment are SAAM and ATAM [6], which will prove whether software architecture is modifiable, what kind of modifications the system is likely to undergo and how the architecture can manage with this changes. Other approaches to evaluate NFRs exist as well, e.g., reliability modeling or the automatic requirement analysis tool [7].

The challenge for evaluating non-functional requirements during early development phases is to evaluate the product, process and/or resources during early phases in order to be able to give an accurate prediction of end-systems quality. On the one side the software cannot properly evaluated unless the requirements are at least known (what will be our product?) and also stable. Therefore evaluation is normally done on end products or at least on intermediate product components that have a certain maturity degree. On the other side market pressure urges companies to do earlier evaluation, as the introduction has shown already. Figure 1 depicts this dilemma of when to evaluate: starting an evaluation having enough data and criteria to judge or starting earlier to be in time to change the process and improve the product.

Looking at the evaluation approaches presented before, it is obvious that not all approaches are able to cover evaluation during the complete life cycle of the software system. For example the code metric approach can only be used when code is available. So during the requirements analysis and design phase the approach is not applicable. Meanwhile, architecture assessment is restricted to design and cannot be used later on in the development process. This means that the development phase determines the evaluation approach that is to be applied. This has to do with our ability to define criteria during the phases of a product.

The critical issue for successful evaluation is to have insight in the criteria and being able with this reference to judge about what is good and wrong about the evaluation object or at least to determine if the thing that is being evaluated is acceptable or not. Such criteria are available for development processes; think about the base practices stated in ISO 15504 or the key process areas defined in the CMM. Based upon these reference models, process assessments, like BOOTSTRAP can be done. These assessments focus on the process and resource issues.



**Fig. 1** The trade-off between having information to ground the evaluation and being in time to change the process and improve the product

We are less optimistic when looking at the criteria or reference models for product quality. Often these criteria are context dependent and not as that stable. This is illustrated in figure 1 with the transfer from the situation where we should deal with unstable requirements and where there are many unknowns to the situation where requirements are stable, with technical certainty. Of course we can partially rely on process assessment, because of the assumption that process quality determines product quality. However, for reliable and mature evaluation of NFRs this approach is too weak. Therefore we need an approach that specifically addresses our needs for evaluating NFRs and that covers process, product as well as the subcontractor perspective on the criteria that are set for the software intensive system. Quality models –that contain these criteria and that are in fact the reference model- plays a crucial role in our approach.

### 3. Quality Models to Evaluate Non-Functional Requirements

This section defines requirements, which quality models should fulfill in order to be able to evaluate NFRs of software-intensive systems successful.

According to [8] there are two kinds of approaches to modeling product quality presented by researchers: fixed-model and define-your-own-model. The fixed-model approach assumes that all important quality characteristics are a subset of those in a published model. To control and measure each quality attribute, the models associated internal quality characteristics, measures, and relationships are used. This contrasts a

define-your-own-model approach where not a specific set of quality characteristics is specified, but rather – in cooperation with the user - a consensus on relevant quality characteristics is defined for a particular system. These characteristics are then decomposed (possibly guided by an existing quality model) to measurable quality attributes result. Examples of the fixed model approach are the models proposed by ISO9126 [9], Boehm [8], Barbacci [10], Dromey [11], Grady [8]. Examples of a define-your-own-model approach are the Goal-Question-Metric (GQM) [12], Factor-Criteria-Metric (FCM) [8], Objective-Principle-Attribute (OPA) [13].

Despite the variety of existing quality models, they do not cover all the important aspects of quality modeling and evaluation. Many of them just replicate the lacks of others. For example fixed quality models define a constant set of quality characteristics. Nevertheless it is unrealistic to assume that it is possible to define a prescriptive view of necessary and sufficient quality characteristics to describe quality requirements for every project. Evaluating NFRs of evolutionary software-intensive systems has guided us to set other requirements to quality models as well. These requirements for quality models are:

- transparent and applicable with little data
- flexible as well as reusable
- applicable for measurement

*Transparent and applicable with little data* - A quality model should be transparent to know the rationale of how certain attributes are related to others and how to subdivide them into sub attributes. For example, the model presented by Barbacci [10] does include software safety aspects while the ISO 9126 [9] does not. Both models do lack a rationale for deciding which quality attributes relate to each other or how to refine an attribute into sub-attributes. As a consequence, selection of attributes, sub-attributes and metrics can be seen arbitrary. Transparency of a quality model does also mean that the meaning of the attributes and their interrelations are clearly (unambiguously) defined. In many models, the distinction between certain quality characteristics according to their definitions is not clear. For example, the average developer will not be able to distinguish between attributes like interoperability, adaptability, and configurability, e.g., in [9], as they might regard them as being identical. Transparency of a quality model requires also insight in interrelationships between the attributes to detect conflicts, trade-offs and redundancies. Those interrelationships are especially important as far as the model accuracy and its predictive capabilities are concerned.

Because of the lack of transparency such approaches as artificial intelligence (e.g., neural networks) and regression modeling –see e.g., [5]- are not suitable for our work on evaluating NFRs. They are in fact “black-box” approaches that do not provide insight into the quality model as well as that their creation cannot be influenced. Another reason why these approaches are excluded is their incapability to cope with little, imperfect data. Artificial intelligence and regression modelling require normally a lot of quantitative data that are often not available in NFR evaluation practice.

*Flexible as well as reusable* - A quality model should be flexible because of the context dependency of software quality. Normally this is denoted by the famous sentence ‘quality is in the eyes of its beholders’, which stresses that different stakeholders in-

volved in the system development have different perspectives on the quality that should be the result. It is widely accepted that stakeholder views –like quality assurance, (sub) contractor or engineer- should take into account to specify quality successfully; see e.g., [14]. The need for flexibility in quality models is also given by software system’s domain and the phase in the life cycle during which the system is evaluated. A project domain considers the different types of software systems, like embedded systems or web applications, which each has their own requirements. The life cycle phase during which a system is evaluated is another dimension that requires flexible quality models. This was already addressed in section 2 where it was made clear that not all evaluation approaches are able to cover evaluation during the complete life cycle of the software system. Methods like ATAM are applied in early stages of systems and software development. The methods as such do not deliver a set of criteria like quality models intend to do. So dependent on the data available during those phases, we can specify the required approach for such phases.

Flexibility of a quality models implies that such models have to be tailored to organization-specific characteristics; therefore the models should be transparent. In addition a quality model should reflect project individual characteristics, namely the stakeholders and lifecycle. At the same time it should be applicable in evolutionary environments and allow learning across similar projects. Fixed quality frameworks – as we referred to in the beginning of this section – do not provide guidance for this tailoring. Despite, define-your-own-models allow an organization custom-tailoring a quality model. Especially the GQM approach is suitable to specify a quality model that matches company-specific needs. However, as such approaches have the disadvantage that they each time start their new measurement program from scratch. Thus they are often not feasible when evaluating evolutionary software-intensive systems, because evaluation requires a considerable amount of effort, time and expertise that requires avoiding this ‘reinventing of wheels’. Therefore quality models should be flexible as well as that they should reuse existing experiences that are stored in existing quality models.

*Applicable for measurement* - The term ‘systematic examination’ that was used to define evaluation in section 2 implies that evaluation should be measurement-based. Measurement is defined as the assigning of values (which may be a number or category) from a scale to an attribute of an entity [9]. Especially in a context of evaluation software-intensive systems it is important that quality models express the entities as well as their attributes. Quality models are inadequate for measurement when they only denote the attributes - the ‘ilities’- of the system and abstract too much from the entities that relate to those attributes. For example, analyzability of documentation might require another evaluation than analyzability of the source code in programmable logic controllers (PLCs).

Some of the quality models; e.g., [9], have restricted accuracy due to fact that they do not explicitly map quality attributes to metrics. There is often no description of how the lowest-level metrics are composed into an overall assessment of higher-level NFRs. Then there is no means to verify how a chosen metric affects the observed behaviour of a quality attribute.

Different distinctions to identify software measurement, like direct versus indirect measurement and internal versus external quality, have their impact on the way attribute relationships should be constructed and the way how the associated metrics, that measure the value of the attributes, should be derived; see [9]. Two opposite types of metrics can be distinguished in evaluation [16], namely: ‘hard’ metrics (like code metrics, see section 2) versus ‘soft’ metrics (like for example applied in a expert judgement or a process assessment). Between those opposites, intermediary forms exist. Next table provides an overview of the factors that determine those two types of metrics.

**Tab. 1** Identification of hard and soft metrics [16]

	‘Hard’ metrics	‘Soft’ metrics
Value (on a scale)	Quantitative	Qualitative
Value assignment	Direct, indirect	Indirect
Attributes	Intern	Extern, Quality-in-use
Measurement procedure	Objective	Subjective

Hard metrics are often preferred above the soft ones because the hard ones are considered as more objective and therefore better. The specific problem with soft metrics is the variance that is generated when doing qualitative measurement. For example, Andersen and Kyster report in [7] about a 40% variance between two expert ratings of the same software product. It is often addressed as ‘subjective’ and therefore avoided. Determining opinions in a systematic way e.g., by using structured questionnaires and closed answer options – a fixed set of response possibilities for each question – can reduce variance. Other possibilities are to apply control questions, repeating questions by formulating them in another way or repeating questions by repeating the response possibilities; see e.g., [17].

An advantage of soft metrics is that they normally are able to measure attributes on a higher abstraction level and that it is often easier to implement them quickly than for example code metrics which face several problems e.g., they are dependent on the programming language. Soft metrics are also often more suitable to measure process as well as resource issues. This makes it necessary to try to apply hard and soft metrics in combination in order to be able to address (intermediate) products (e.g., size, reliability, maintainability) as well as process (e.g., effectiveness, efficiency) and resource issues (e.g., capability, capacity).

#### 4. Prometheus: A Probabilistic Quality Model

This section presents the Prometheus approach. Prometheus stands for Probabilistic Method for early evaluation of NFRs. Prometheus develops quality models, which fulfill the requirements that were discussed in previous section. In order to achieve this we have combined three existing methodologies to modeling and evaluating quality of software product: Software Quality in Development (SQUID), Bayesian Belief Networks (BBN) and Goal-Question-Metric (GQM).

We apply concepts of the *SQUID approach* [15] to specify quality attributes and their dependencies in a measurement-based way. It also addresses the transparency, reusability and flexibility requirements for a quality model.

A structure quality model is drawn that allows us either adopting one (or more) of the existing quality models or developing from scratch the new model that best suits organization's characteristics. The basic elements of the structure model are: entity, attribute, value, unit and measurement instrument.

An *entity* can be any (intermediate) product or artefact, like document or module source code. It might be also a process (like requirements engineering or configuration management) or a resource (like a review technique, supporting tool or the development team). An entity possesses one or many *attributes* while an attribute can quantify one or many different entities. During an evaluation we want to determine the value of the attributes. Therefore, measurement is conducted during which values are assigned to the attributes. The *value* is than what is obtained by applying a specific measurement *unit* to a particular entity and attribute.

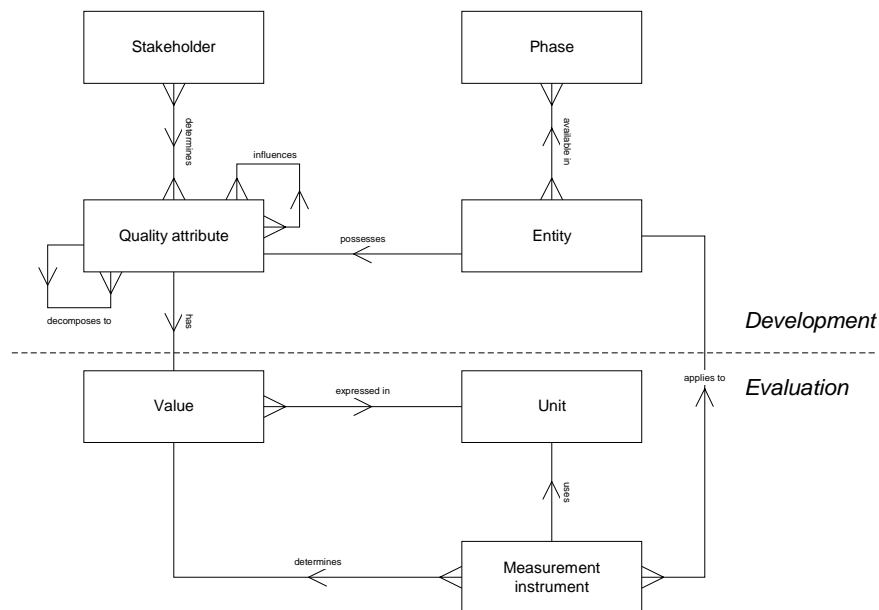
The metrics that are used for such measurement are represented in the structure model by unit. A *metric* is the unit (or scale) with the procedure to determine the value. The use of units is extended beyond the ratio and interval scales in order to allow for the definition of the scale points for the ordinal scale measures and categories used for nominal scale measures. This supports distinguishing between soft and hard metrics.

The technique that helps us with doing measurement is defined in the structure model as *measurement instrument*. This might be the shell to determine the value of the code metrics to be able to start a code analysis. With representing the element measurement instrument in the structure model, it is possible to distinguish between the measurement level (represented by unit and value) that is needed for the logical description of metrics and the technique level that we need for implementing those metrics during an evaluation.

We propose to add the elements stakeholder and phase to the existing SQUID model. The relevance of stakeholder has been discussed in the previous section. By modeling stakeholder in a relation to the element attribute, different views on quality could be reflected in different structure and content of a quality model. The importance of *phase* in the software life cycle is already elaborated in section 2 and 3. The phase is related to entity because different product artefacts become available during the development life cycles as well as different processes are conducted and different resources are applied during subsequent phases. With acknowledging phase in the structure model it will be possible to determine which entities are possible to evaluate at which point of time.

Another change that we propose in addition to SQUID concerns the decomposition of attributes. The SQUID approach distinguishes two types of decomposition, namely: the decomposition from quality attributes into sub-characteristics that refines different aspects of the original characteristics (like ISO 9126) as well as decomposition into criteria related to properties that are believed to influence achievement of quality attributes requirements (like the McCall model). Both relationships are important, but

we think that it is better, to express the elements ‘internal software property’, ‘quality characteristic’ and ‘quality sub-characteristic’ not separately like SQUID does. The problem with this way of representing is that it will be often difficult to reuse such abstractions made in particular context for another context, because of their specific meanings. Therefore we propose a simpler representation approach and define only the element attribute with two additional relationships, namely: decompose and influence. Below our structure model – which is an adaptation of the SQUID structure model – is presented as an Entity Relationship diagram.



**Fig. 2** Prometheus: the structure model.

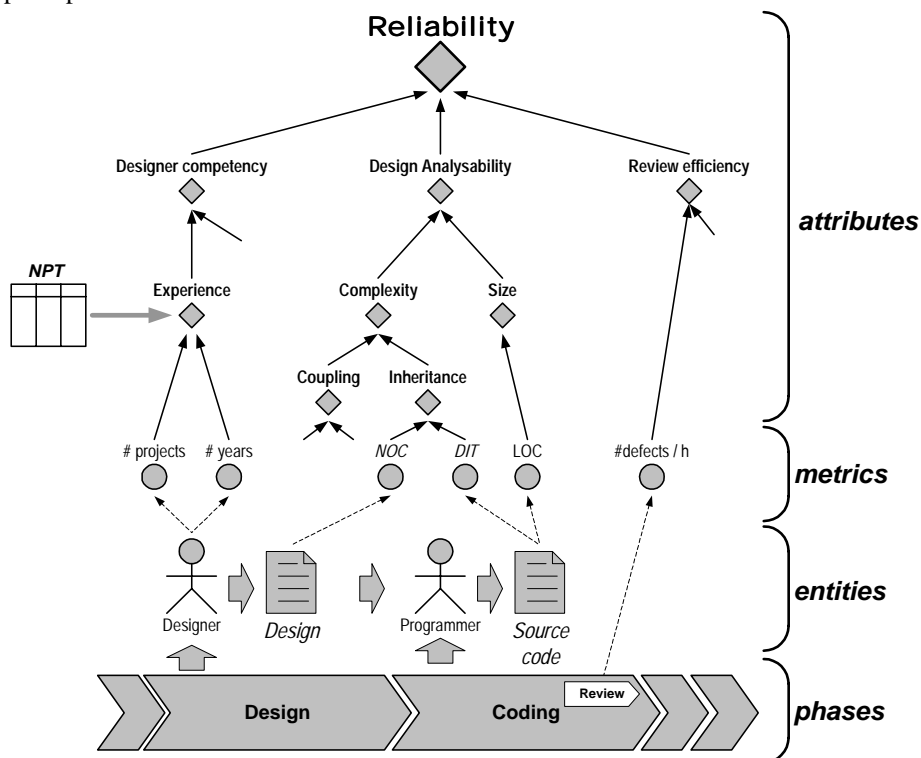
This structure model is used to derive actual decomposition trees, see the example in figure 3. In section 5 we will show that GQM-techniques like the goal measurement templates and abstraction sheets can support us in achieving the decompositions trees.

The *BBN approach* [18] is applied in addition to the structure in order to be able to model the decomposition and influence relationship among quality attributes and to reason (basis on those relationships and measurement data) about NFRs. With applying the BBN we address the following requirements for quality models: their transparency, reusability, dealing with missing data and different views (flexibility).

A BBN is a graphical network, which has a similar structure as the quality model (structure as well as content), contains a set of probability tables, which together represent probabilistic relationships among variables (attributes). The BBN graph consists of nodes and arcs where the nodes represent uncertain variables (discrete or continuous) and the arcs the influence relationships between the variables. Each node is

described with a Node Probability Table (NPT), which cover conditional probabilities of the node's state basis on the states of nodes that influence this node.

An example of how to apply BNN in combination with the quality model is provided below given. The quality model that is presented in figure 3 is an example of a BBN for "reliability prediction" problem, which includes also the NPTs. The nodes represent attributes (discrete or continuous), for example, the node "Experience" is discrete and can have three values "low", "medium" and "high" whereas the node "size" might be continuous (such as LOC). Dependent of the organizations capabilities and needs it is of course possible to represent as discrete the variables which are in principle continuous.



**Fig. 3** Example of quality model developed with the use of Prometheus

For example we might represent size of code as "small", "medium" and "large". The arcs represent influential relationships between attributes. The number of similar projects he/she participated in and years of experience define designer experience. The exemplary node probability table (NPT) for "Experience" node might look the one shown in table 2. For the simplicity of the example we have made all considered in NPT nodes discrete so that each of them take on just three discrete values. The NPTs capture the conditional probabilities of a "Experience" node based on given the state of its parent nodes.

As we already mentioned there may be several ways of determining the probabilities for the NPTs. One of the benefits of BBNs stems from the fact that we are able to

accommodate both subjective probabilities (elicited from domain experts) and probabilities based on objective data.

**Tab. 2** Example of Node Probability Table

# similar projects		few			many			lot		
years of experience		few	many	lot	few	many	lot	few	many	lot
experience	Low	0,70	0,50	0,33	0,50	0,33	0,20	0,20	0,33	0,70
	Med	0,20	0,30	0,33	0,30	0,33	0,30	0,30	0,33	0,20
	High	0,10	0,20	0,33	0,20	0,33	0,50	0,50	0,33	0,10

Having entered the probabilities we can use Bayesian probabilities to do various types of analysis. The most interesting is the propagation i.e. updating the actual probabilities after filling the model with evidence (fact). For example measured actual values of “# of similar projects” and “years of experience” are facts, and could be propagated in order to update the probabilities of “Designer level of experience”. For detailed look at how BBN perform propagation we refer to the BBN-related literature e.g., [20].

In principle, the exemplary model represents developer and maintainer view on the reliability requirement. It is also possible to merge more than one view in one model. Adding new attribute nodes related to another stakeholder could do it. In case that stakeholders differ with regard to probabilities we can either combine their assessment (e.g., average) or in large discrepancies create a new instantiation of the model and feed it with different probabilities (NPTs).

The benefit of applying BBN (the NPTs) in combination with the quality models is its ability to reason under uncertainty and to combine the advantages of an intuitive visual representation with a sound mathematical basis in Bayesian probability. This allows us to articulate information about the dependencies between different variables and propagating consistently the impact of evidence on the probabilities of uncertain outcomes, such as “product reliability”. The dependency information could be either the result of theoretical proves, empirical studies or just expert beliefs. It means that we can deal with subjectively or objectively derived probability distributions. Other benefits of BBNs are:

- (easier) specifying quality attributes and understanding of modeled relationships among them (contradictions, redundancies) via transparent, graphical format;
- refining probabilities (across subsequent development phases or projects) in order to increase the accuracy of prediction;
- predicting with missing data;
- automatic tool support [18].

## 5. Specifying an Evaluation

Previous section has described the Prometheus approach to store information necessary to define transparent, flexible and reusable as well measurement quality models. This section focuses how to store the information in quality models and how to get information out of it during the evaluation process.

Based upon the general process for doing software product assessments [21] and the Quality Improvement Paradigm [22] the following phases for an evaluation process are distinguished:

- specify the evaluation of NFRs – this is to specify the non-functional requirements about a particular software-intensive system into measurable terms. The result of the specification phase will be an evaluation plan;
- execute the evaluation – this is the actual evaluation to determine the fulfillment of the system to its specified NFRs. This phase is conducted according to the evaluation plan and applies the evaluation techniques;
- define actions – this is definition of the actions to improve the product or process based upon the results from the execution of the evaluation. The actual execution of these actions, like e.g., starting reviews or component-based development is very important for using the evaluation results, but is not part of the evaluation. The definition of those actions should be considered as a phase in the evaluation;
- package evaluation experiences – this phase is important from an evaluator's perspective because it should provide the evaluator with information about criteria, techniques that might be applicable for future evaluations.

The remainder of this paragraph focuses on the first phase mainly. When specifying the evaluation we are concerned about how to reuse existing quality models. In fact, a match has to be made between the specific NFRs of the system that is under evaluation and the attributes (plus associated metrics and measurement instruments) proposed by the quality model. The goal measurement template and the abstraction sheet are both techniques that facilitate this matching. Both techniques originate from the Goal Question Metric (GQM) method [12]. The goal measurement template helps in specifying the expectations about the evaluation. The template addresses five elements, namely:

- object – what will be analyzed? (e.g., processes, products or resource);
- purpose – why will the object be analyzed? (e.g., characterization, evaluation, monitoring);  
prediction, control, improvement [12]
- quality focus – what property of the object will be analyzed? (e.g., cost, correctness, defect removal, changes, reliability, user friendliness, maintainability);
- viewpoint – Who will use the data collected? (e.g., user, senior manager, project manager, developer, system tester, quality assurance manager);
- context – In which environment does the analysis take place? (e.g., organization, project, problem, processes, etcetera).

All elements of the template are equally important in the process to specify a specific evaluation plan, but especially the object and quality focus are relevant for reuse, because they refer to entity and attribute in the quality model respectively. Having defined the object and the quality focus for a particular evaluation, it specifies what we need from the quality model. The viewpoint in the measurement goal template defines which stakeholders should be taken into account during the evaluation. This matches to the element Stakeholder in Prometheus.

Abstraction sheet is an additional technique that will help in further specifying what is to be reused from the quality model. An *abstraction sheet* summarizes the main issues and dependencies of an evaluation goal on four quadrants in a page, namely:

- quality model – the quality factors: what are the measured properties of the object in the goal?
- baseline hypothesis – what is the current knowledge with respect to the measured properties? The level of knowledge is expressed by baseline hypotheses.
- variation factors – which factors are expected to have an impact on quality models?
- impact on baseline hypothesis – how do these variation factors influence the quality model? What kind of dependence is assumed?

The abstraction sheets might already contain information from the quality model. When doing this it will be feasible to propose quality factors to the stakeholders so that a discussion can start on their relevance for the systems that has to be evaluated.

The goal measurement templates and abstraction sheets can also be applied to package the experience into a new quality model. The factors depicted in the abstraction sheets and the quality focus in the templates will transfer into the attributes, while the viewpoints and objects of the template are associated with the stakeholder and entity respectively.

Having specified the attributes and metrics for the evaluation the information the evaluation techniques have to be selected and assembled. The selection concerns the choice of appropriate techniques that can evaluate the stated attribute and that can cope as well with the output that results from the specified life cycle phases. General descriptions are needed that define the expected outputs per life cycle phase, for example that the requirements analysis phase is related to e.g., requirements, change requests, problem reports. Similar information should be gathered for the respective evaluation techniques, so that they can be selected.

The technique assembly is about putting the set of techniques in an order so that they can be applied for efficient and effective data collection. For example, in case of applying different assessment questionnaires the assembly will deal about when and how to consult the engineers to avoid too much disturbances. The assembly is also about passing the results of a technique to its successors, e.g., to be able to trace the same attribute during several phases of the life cycle. How to set up a taxonomy for information necessary for selection and assembly is subject of further research.

The selected evaluation techniques are to be assembled (constructed) into an integrated approach. The integrated approach makes it possible to collect information from the software process as it become available during the software engineering work and makes evaluation possible at selected intervals, such as project milestones.

## Conclusions

In this paper we have argued that insufficient applicable criteria are available to evaluate non-functional requirements (NFRs) of evolutionary software-intensive systems during their early development phases. Therefore we need quality models that are transparent (for their rationale as well as the meaning of the attributes), able to deal with little data, flexible (being able to be tailored), reusable and measurement-based (which implies the application of hard as well as soft metrics).

An approach, called Prometheus, was introduced to cope with these requirements. It is basically a way of modeling quality that is measurement-based, that links the attributes to the evaluation objects, which might be processes, intermediate products as well as resources. The relationships between the attributes are modeled with probability tables. The quality models will be applied during a goal-oriented specification of an evaluation. The results of particular evaluation should be packaged in the quality models.

Our approach is particularly interesting for projects and organizations that want to perform measurement-based evaluation in a context of evolutionary systems. Prometheus enables them to learn effectively over several product variances/releases and will refine the proposed quality model through subsequent projects. The approach is meant to be applied during demonstrator projects in the ITEA EMPRESS project.

## References

1. Elixmann, M. and St. Hauptmann, Software Maintenance on the fly, in: Philips Res. Bull. On Software and Systems, no.14, November 1994
2. Belady, L.A., M.M. Lehman, A model of large program development, IBM Systems Journal, no. 3, pp.225-251, 1976
3. Minderhoud, S., Quality and reliability in product creation – extending the traditional approach, in: Quality and Reliability Engineering International, December 1999.
4. Bahill, A.T. and Dean, F., "Discovering system requirements", Chapter 4 in the Handbook of Systems Engineering and Management, A.P. Sage and W.B. Rouse (Eds), John Wiley & Sons, 175-220, 1999
5. Briand, L., J. Wuest, "Empirical Studies of Quality Models in Object-Oriented Systems", to be published in Advances in Computers, Academic Press, updated Feb. 18, 2002.
6. Kazman, R., e.a., The Architecture Tradeoff Analysis Method, in: 4th International Conference on Engineering of Complex Computer Systems, augustus 1998.
7. N.E. Fenton S.LO.Pfleeger, "Software Metrics: A Rigorous and Practical Approach", PWS ISBN (0534-95429-1), 1998 (originally published by International Thomson Computer Press, 1996)
8. Bache and Bazzana, Software metrics for product assessment, London, McGraw-Hill Book Company, 1994
9. ISO/IEC 9126 International Standard, Software engineering – Product quality, Part 1: Quality model, 2001
10. M. R. Barbacci, M. H. Klein, T. Longstaff and C. Weinstock, C. "Quality Attributes", Technical Report CMU/SEI-95-TR-021, ADA307888. Pittsburgh, PA, Software Engineering Institute, Carnegie Mellon University, December 1995.

11. Dromey, G., Cornering the Chimera, in: IEEE Software, January, pp.33-43, 1996.
12. Solingen, R. van, E. Berghout, The Goal/Question/Metric method – a practical guide for quality improvement of software development, London, McGraw-Hill, 1999.
13. R.E. Nance, J.D. Arthur, “Managing Software Quality, A Measurement Framework for Assessment and Prediction”, Springer 2002
14. Kusters, R., R. van Solingen, J. Trienekens, H. Wijnands, ‘User-perceptions Of Embedded Software Reliability’, Proceedings of the 3rd ENCRESS Conference 1997, Chapman and Hall, ISBN 0412802805, 1997
15. Kitchenham, B., S. Linkman, A. Paquini, V. Nanni, The SQUID approach to defining a quality model, in: Software Quality Journal, Vol.6, pp.211-233, 1997.
16. Punter, T. , Goal-oriented evaluation of software products (in Dutch), PhD thesis, Eindhoven University of Technology, 2001.
17. Xenos, M., D. Stavrinoudis, D. Christostodoulakis, *The correlation between developer-oriented and user-oriented software quality measurements*, in: Proceedings of International Conference on Software Quality, Dublin, pp.267-275, 1996.
18. Fenton NE, Krause P, Neil M, "A Probabilistic Model for Software Defect Prediction", accepted for publication IEEE Trans Software Eng, Sept 2001
19. J. Pearl, “Probabilistic Reasoning in Intelligent Systems: networks of plausible inference”, Morgan Kaufman 1988
20. [http://www.hugin.com/Products\\_Services/](http://www.hugin.com/Products_Services/)
21. ISO 14598-1, Information technology – Software product evaluation, Part 1 – General overview, Genève, ISO/IEC, 1999.
22. Basili, V.R., C. Caldiera, H.D. Rombach, Experience factory, in: Encyclopedia of Software Engineering, J. Marcianiak (ed), Vol.1, John Wiley and Sonns, pp. 469-476, 1994.