A Systems Perspective on the Quality Description of Software Components[1]

Otto Preiss
ABB Switzerland Ltd, Corporate Research
5405 Dättwil, Switzerland

and

Alain Wegmann
School of Computer and Communications Sciences, Swiss Federal Institute of Technology
1015 Lausanne, Switzerland

ABSTRACT

In this paper we present our rational for proposing a conceptual model for the description of quality attributes of software artifacts, in particular suited to software components. The scientific foundations for our quality description model are derived from researching systems science for its value to software engineering. In this work we realized that software engineering is concerned with a number of interrelated conceptual as well as concrete systems. Each of them exhibits the basic system theoretic principles and is strongly related to certain types of qualities. Such qualities receive particular attention in the context of large software systems, where systems are a combination of in-house and third party products and are increasingly integrated by means of software component technology. Consequently, a quality data sheet is needed by component users to gain trust in, and to evaluate the possible employment of, a candidate component. Interestingly, the concept of a software component appears in most of the aforementioned different types of systems. Hence, it is an excellent means to carry quality related information that belonged to different spheres up to now. The qualities range from those related to the development economics to those related to the execution performance.

Keywords: Systems science, software engineering, software components, quality attributes, non-functional properties, quality model.

1. INTRODUCTION

While, with today's techniques, methods and tools, we seem to be able to build any system and its required operation somehow and sometime, the big challenge lies in building such systems in a repeatable manner with predictable qualities of different kinds. These qualities range from those related to the development economics to those related to the execution performance. In general, they refer to the large group of properties that are sometimes referred to as Ilities [1], non-functional properties (in most of the literature), afunctional qualities [2], extra-functional properties [3, 4], or simply quality attributes. While some literature carefully distinguishes between these terms, other sources seem to use them almost synonymously. A common understanding is that one class of quality attributes is observable at software system runtime (such as dependability, usability, security, etc.) while another is not. This latter class relates to the economics of building and evolving a software system and its artifacts and is focused on shortening time-to-market and on decreasing development, maintenance, and non-conformance costs. Hence, it deals with quality attributes that are observable over the product lifecycle (such as maintainability, reusability, etc.). A relatively young concept is that of a software component [5]. As we argue in [6], the "raison d'être" for software components is to deal with quality attributes of both basic types.

The software community's current means to cope with quality attributes is largely based on technology (such as software component technology), on experience, and on codified best-practice information. Be it in the area of development processes [7] or in the area of architecture, design and implementation [8] [9]. While access to such an empirical body of best-practice knowledge is useful for a prospective design of a new system, we are challenged by the question of whether there is an underlying big picture, a conceptual framework or unified view that would allow us to understand quality attributes in a bigger context. What makes it so hard? We believe that there are two sorts of complexity sources – the many systems related to software systems and the abstractness of many of the quality concepts. More concretely, we see the following obstacles:

- Computers, like brains, are intelligent, complex symbol processing systems [10].
- The artifacts (the symbols) that are fed into a computer system, usually in the form of some executable or source code, are models in the form of conceptual systems.
- The environment surrounding a computer system is itself a complex system consisting of a number of complex systems.
- The software development team and its members are complex systems.
- We tend to use different models, different views, and different representations for systems to facilitate our understanding. However, correspondences between entities in these different understanding aids become increasingly hard to establish and trace.
- Many of the quality attributes (e.g. maintainability) are too abstract to be ascribed to a thing. They will only become

[1] This is an extended version of the paper presented at the 6th World Multiconference on Systemics, Cybernetics and Informatics (SCI2002), Orlando, July 14-18, 2002, Vol. VII, p. 250-255.

tangible if brought into a specific system context and related to behavior, i.e. related to some form of interaction with the thing.
- There is no natural distinction between a functional and non-functional quality.

Consequently, as a first step, we wanted to better understand the phenomena of quality attributes, and their relationship to systems and their organization. For this, we looked into systems science and tried to apply some of the findings, which we document in the rest of the paper. In section 2, the heart of the paper, we discuss systems science and our findings which are relevant for this paper. In section 3, we briefly motivate our decision to concentrate on software components as the main carrier of quality related information and we present the basic structure for a quality description model that is based on the insights gained from our understanding of systems science. In section 4, we summarize this paper and hint at the applicability of such a model.

2. THE VALUE OF SYSTEMS SCIENCE

Elsewhere [11] we presented some of the basic principles of systems science and our interpretations. In essence, we concluded that software systems are complex systems that can benefit from systems science as originally founded by Ludwig Van Bertalanffy in what he called General System Theory [12]. With respect to this paper, system science was useful (a) to derive our so-called "2-2-2 model" and (b) to understand the value of looking at software engineering as a set of different types of systems.

Before we discuss our application of system theoretical principles, we cite a definition for system that we found generic enough to be applicable in all circumstances, yet concrete enough to be useful:

"A system is a set of interacting units with relationships among them. The word "set" implies that the units have some common properties. These common properties are essential if the units are to interact or have relationships. The state of each unit is constrained by, conditioned by, or dependent on the state of other units. The units are coupled. Moreover, there is at least one measure of the sum of its units which is larger than the sum of that measure of its units." [13]

The 2-2-2 Model

"2-2-2" refers to our conceptual world-view which relies on the pattern: *two systems, two views, and two domains of inquiry.* The following reasons motivate this pattern:

(1) *Two systems:* Complex systems are hierarchical systems [10, 13]. A system cannot be modeled without considering the system it is embedded in. Consequently, there are always at least two systems to be considered: the *suprasystem* and the *system of interest* (SoI). This is not new, of course. Checkland termed this the "twin-systems", in the sense that we cannot model a serving system if we do not know what is being served [14]. Simon used the concepts of an inner and an outer environment to describe the same circumstances [10].

(2) *Two views:* We must always explicitly consider both the properties as perceived by outsiders of a system and the mechanisms that yield these properties, i.e. the inside view. Because of an engineer's focus on synthesizing an object with desired properties, we call these separate views "the goals of a system" vs. "the means to achieve the goals".

(3) *Life cycle based domains of inquiry*: We must apply the system science principles in a functional context. What software engineers call the software system life cycle lends itself to define the relevant domains of inquiry. Hence, we pick the development context and the execution context of a software system, which we call operational context.

Accordingly, the 2-2-2 model depicted in Fig. 2-1 consists of three dimensions with two distinct points of reference per dimension. Table 2-1 gives an explanation of the reference points per dimension. The conceivable eight intersections represent meaningful perspectives for modeling. Each perspective can and shall be analyzed because it addresses a special set of concerns, i.e. it is useful to state questions and elaborate on answers that are valid to some class of stakeholders. Consequently, we have shown the applicability of this conceptual framework to classify the wide variety of stakeholders [11] and the applicability of the basic systemic principles in an enterprise system architecture methodology called SEAM [15].

To quickly related this framework to the everyday situation of a software engineer, let us give concrete instances of suprasystems and SoI for the development and operational context, respectively:

- Suprasystem in development context: The development company (with its units being the departments, the development projects, etc.).
- SoI in development context: The development project (with its units being the people, tools, repositories, etc.). The output of that system is the developed artifacts.

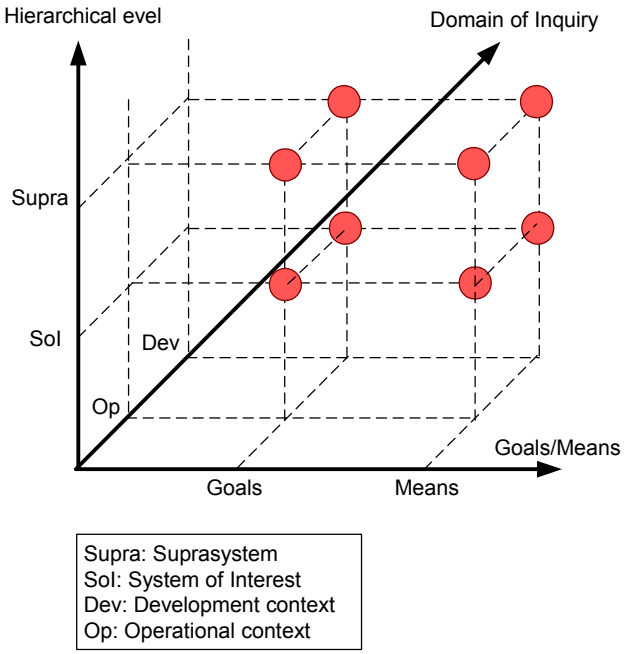

Figure 2-1: The 2-2-2 Model

- Supra-system in operational context: The company that is using the computer system with its the application

software that was developed by the development company.
- SoI in operational context: The computer system with its application software at runtime, i.e. the developed software executing in its deployed environment.

Table 2-1: Informal definition of framework dimensions and reference points

Dimension	Points of reference	
Goals/Means	*Goals*; concerned only with the perceived (required) behavior of a system (the internals of which are not relevant or even unknown to the interested observer).	*Means*; concerned with the internals of the system, i.e. the structures and processes that provide the total perceived behavior.
Hierarchical Level	*SoI;* one particular system of further interest out of the set of systems in the supra-system.	*Supra*; the organizationally[2] higher-level system, i.e. the set of systems including the SoI, again seen as a system.
Domain of inquiry[3]	*Dev*; the life cycle phase (a time/space construct) that constitutes the conception and the design of the envisioned (future) system.	*Op*; the life cycle phase that stands for the execution of a system in its operational environment.

The framework proves valuable not only to classify stakeholders but also to position the wide range of quality attributes and their relationships, as well as support stakeholder traceability for these attributes. For instance, let us consider *time-to-market* as an attribute relevant to the goals viewpoint of the suprasystem in the development context, i.e. it is a goal of the development company. One possible means for this goal could be *reuse* of source code artifacts across development projects. In this case, reuse belongs to the means viewpoint of the suprasystem in the development context. Next, one goal for the SoI in the development context, i.e. for a particular development project, is to support reuse by realizing pieces of functionality in the form of *modular* source code units. The means to achieve this modularity goal is to maximize module *cohesion*, minimize module *coupling*, etc. Thus, the causality chain or goals decomposition, where also individual sub-goals are relevant for certain stakeholders, is:
time-to-market → {reusability,…} → {modularity, …} → {cohesiveness, coupling, …}.
Frameworks and tools to support non-functional goal decompositions are available [16].

[2] In his living systems theory, Miller [13] identifies seven organizational levels.

[3] To make our point, we can limit ourselves to two life-cycle phases: creation and operation. A finer granularity is always possible, however.

The Different Systems

General system theory proposes a number of system categories or taxonomies [13] [12]. However, it seems agreed that there are two basic types of systems (Figure 2-2): *conceptual systems* (also called symbolic or symbol systems) and *concrete systems*. While a conceptual system is always a designed system (i.e. created by man), a concrete system may be a designed or a natural system, or even a hybrid.

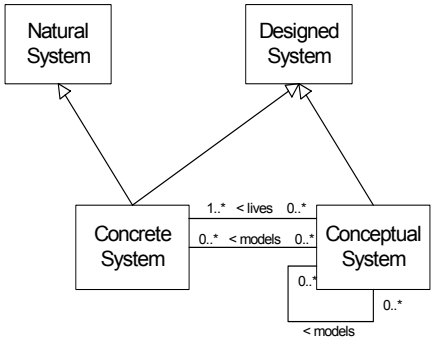

Figure 2-2: Basic Types of Systems

More specifically, concrete systems are a non-random accumulation of matter and energy in physical time/space and their units and relationships are empirically determinable by some operation carried out by an observer. Conceptual systems on the other hand may be purely logical or mathematical. Some sort of formal identity or isomorphism (or more accurately homomorphism) with units and relationships of concrete systems may exist. However, all of the units and relationships of conceptual systems are selected by scientific observers or theorists. Conceptual systems always live in one or more concrete systems. An organism, a human being, a social system, an electronic system (a radio or a computer) are concrete systems. A theory, a language, a computer program, and others are conceptual systems.

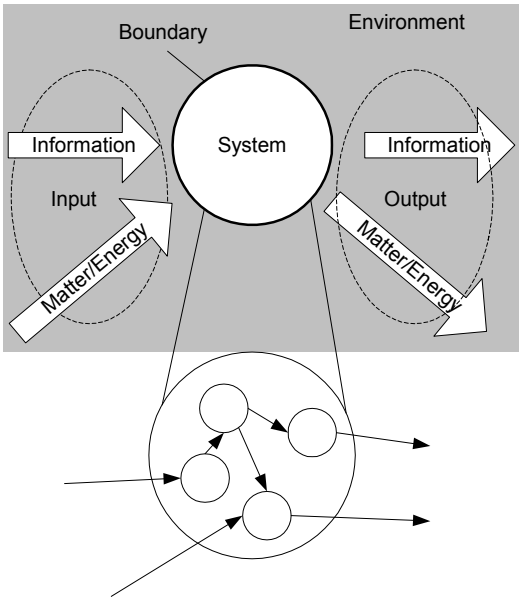

Figure 2-3: Fundamental Concepts of a System

The fundamental concepts for any concrete system are depicted in Figure 2-3. The figure complements a similar one in [17] with our understanding of information and matter/energy as discussed in [13]. The essential message is that the concrete systems of interest to us process matter/energy and information (collectively seen as input and output) and that they are composed of parts (sometimes called subsystems, units, components, etc.), which in turn are systems. Thus, any system is a component of its suprasystem.

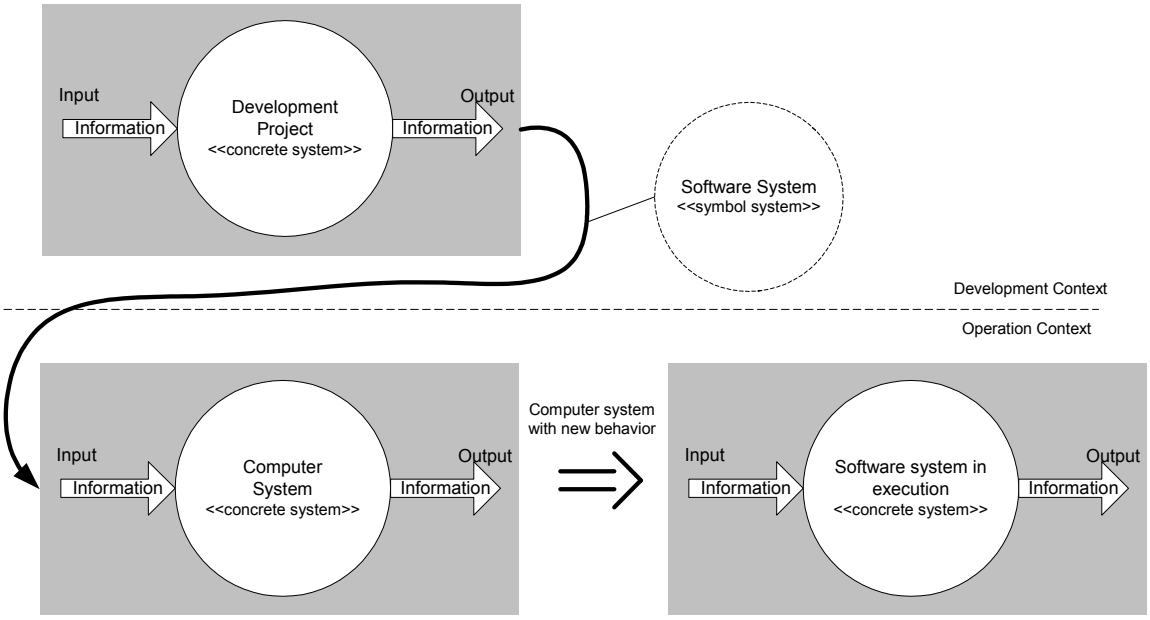

Figure 2-4: The relevant systems for software development

Based on this basic model of a concrete system, on the 2-2-2 model as presented in the previous subsection, and on the awareness of the different types of systems, we infer that for software engineering purpose we need to consider at least three distinct systems of interest of two different types, as is depicted in Figure 2-3:

1. The development project team: It is a concrete system, a human-activity system, that creates the development artefacts, a subset of which is later deployed to the target environment for the software system to execute. In the context of quality, current software engineering terminology refers to qualities related to this system often as the development process qualities.

2. The software system: It is a conceptual or symbol system that is produced by the development project. The observer [13] or intelligent system [10] that processes this symbol system is the computer (or a human when we consider the creation, modification, or inspection of such a symbol system). Current software engineering terminology refers to this system often as the "software product" or as the "system at design time".

3. The software system in execution: It is a concrete system (realized by inputting the software system into the computer system). In fact, the software system is the input information that changes the behaviour of the computer system without the input software to the behaviour expected from the running application. After the input is processed. Current software engineering terminology refers to this system often as the "system at runtime".

3. QUALITY DESCRIPTION MODEL

As a consequence of the above three types of systems, we have to distinguish three basic categories of qualities when we refer to software engineering and quality attributes.

a) Process qualities; these qualities attempt to characterize the software development process and are typically found as criteria in the assessment of the process maturity. Examples are: configuration management usage, review strategies and implementation, documentation habits, etc.

b) Static software product qualities; these qualities attempt to characterize the static qualities of the symbol system. Examples are: all attributes that relate to maintainability or reusability such as complexity, analysability, etc. Since we mostly treat the program (source) code as the relevant input, the software engineering community has developed fairly elaborate metrics to measure and describe static qualities of source code [18]. However, since the systems are conceptual the quality attributes are too. If the future of "programming" were in model-based execution, we would have to redefine and probably invent new metrics for model qualities. We should note that properties (or qualities) in general, but even more so properties of conceptual systems, are inventions of man. I.e., questions about which properties exist are empirical! That is, there

is no a priori or logical method to determine which properties exist [19].

c) Execution qualities; these qualities attempt to characterize the deployed system at runtime. They represent the observable qualities of a software system in a concrete end-user context, i.e. they are essentially behavioural qualities and we can call them qualities of service. Examples are of high-level qualities are: reliability, security, availability, timeliness, etc.

It is interesting to note that the quality model presented in the ISO9126-1:2001 [20] comes fairly close to this basic discovery of three fundamental systems. The standard also bases its parts on a three-phase lifecycle model with process (process quality), software product (internal/external quality attributes), and effect of software product ("quality in use" attributes). However, it is not flexible enough because it restricts the quality characteristics to a few defined ones.

The Case for Software Components

The definition for a system presented in the beginning of section 2 calls for the identification of the relevant units that are part of a system. It is interesting to notice that the concept of a software component is a prominent unit in all of the three systems mentioned above and has the ability to serve as a fulcrum point in linking together these quality worlds. More specifically, a software component is the unit of development, of integration, of reuse, and of versioning in the development project system. Hence, it may carry process related qualities. Secondly, it is the unit of deployment and the unit (or term as it is called in conceptual systems) of which the program code is composed[4]. Hence, it may carry static software product qualities, although the appropriateness of many current source code-related metrics must be re-evaluated for software components. Thirdly, a software component is the unit of execution and of service provision on the target environment. Hence, it may carry execution qualities.

This reasoning led us to our hypothesis that a software component is the one concept that has the ability to carry quality related information that is relevant for different purposes and stakeholders. It can support those that implicitly expect software artifact quality from development process quality, those that expect development economics from the static qualities of software artifacts, and those that are interested in reasoning about runtime assembly-level properties based on software component properties [21].

We should mention that that we base our ideas on those definitions of a software component that define it as an immutable, deployable unit in binary form, which is subject to third party composition, can be put into repositories, and is accessible as well as publishes its properties through interfaces only [5, 22].

Data Sheet for Software Components

Based on the above discussion we propose a model for the logical structure of a quality description of a software component as depicted in Figure 3-1.

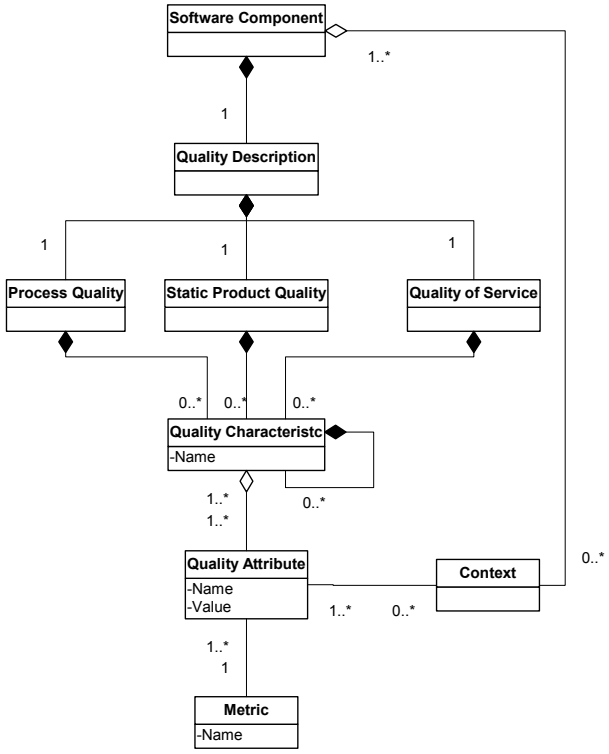

Figure 3-1: A UML metamodel for the Quality Description of a Software Component

We assume that a software component refers to a partial solution of value that is packaged for reuse and can be of interest for a component consumer. Conceptually, it has one quality description that is composed of three sections: *Process Quality*, *Static Product Quality*, and *Quality of Service*. Each section consists of an enumeration of *Quality Characteristics*, each of which is defined by a set of *Quality Attributes*. While quality characteristics are a means to classify qualities (e.g. performance), quality attributes are the tangible concepts that can be assessed by an observer (e.g. latency). Quality attributes relate to one particular *Metric* and they can be associated with one *Value* on some scale out of the range of possible values defined in the metric. A value may of course be a complex structure and accommodate for things such as confidence intervals, tolerance, parameterization, and others. It is important to mention that at least the attributes of the quality of service section, which in essence are behavioral, relate to one or more *Contexts*. A service context is the generic concept to capture the fact that a particular attribute has dependencies. E.g., it may relate a quality attribute to a certain usage pattern. A pattern could for instance represent a permissible method invocation sequence in some form of process algebra, e.g. as a path expression [23], or it could reference a use case description that is available in some form. Very concretely, a quality attribute (say latency) may be specified for certain methods or a sequence of methods only and is thus defined in the context of one specific interface and its one or more provided methods. Further, the context may be used to relate a quality attribute to a rely/guarantee type of construct.

Since this model is only a metamodel it needs to be specialized/extended to be instantiable for a software component at hand. That is, the definition of concrete characteristics, attributes, and metrics for the case at hand is to

[4] It is also a prominent unit in many design models (i.e. conceptual systems) above the program code.

be made. Further, the description of the logical structure is not enough. Pieces of information (such as parts of the descriptions) must of course be available in some electronic form. Typically, these are files residing on machines connected by networks. A second model, the physical structure, could therefore define how the sections and pieces of information that logically belong together are physically arranged. A physical structure can be interpreted as a virtual directory for all pieces related to a component quality description. XML Schemas would of course be suitable to represent the physical structure. A concrete proposal for creating such physical structures are given in [24].

4. CONCLUSION AND FUTURE WORK

In this paper, we have presented an interpretation of some basic principles of systems science to better understand the wide range of quality attributes found in software engineering. Systems science helped to derive the 2-2-2 model and to make explicit the various types of systems that we commonly deal with in the software life-cycle. Developing software with desirable qualities in a repeatable manner is one of the pressing challenges for software development. Systematic software reuse is still the most attractive overall approach to shorten development time, save costs, and improve quality. As opposed to previous reuse attempts, software components are promising because they are the units that are present in all of the most important different types of systems, i.e. they play an central role in the various orthogonal means to improve quality. We hypothesized that software components may be annotated with quality descriptions that reflect process related-, static software product related-, and runtime related quality attributes. Consequently, we sketched a UML metamodel that represents the generic quality description model for software assets, in particular software components. Such a quality model could be a conceivable extension of asset specification frameworks such as the Reusable Asset Specification [25] (especially the component profile) or a basis for works similar to the IEEE Study Group that looks into a mechanism for grading the quality of software component source packages[5] [26].

References

[1] F. Manola, "Providing Systemic Properties (Ilities) and Quality of Service in Component-Based Systems," Object Services and Consulting, Inc., Technical Report, February 1999.

[2] P. C. Clements, "Coming Attractions in Software Architecture," Software Engineering Institute - Carnegie Mellon University, Pittsburgh, Technical Report CMU/SEI-96-TR-008, January 1996.

[3] E. Hochmüller, "Towards the Proper Integration of Extra-Functional Requirements," *The Australian Journal of Information Systems,* vol. 7 (Special Edition - Requirements Engineering), 1999.

[4] M. Shaw, "Truth vs Knowledge: The Difference Between What a Component Does and What We Know It Does," in *Proc. 8th International Workshop on Software Specification and Design (IWSSD-8)*, 1996, pp. 181-185.

[5] In this IEEE draft "software component" refers to any set of source code assets and its related documentation.

[5] C. Szyperski, *Component Software - Beyond Object-Oriented Programming*. Reading, Massachusetts: Addison-Wesley, 1998.

[6] O. Preiss, A. Wegmann, and J. Wong, "On Quality Attribute Based Software Engineering," in *Proc. 27th Euromicro Conference*, 2001, pp. 114-120.

[7] S. Zahran, *Software Process Improvement - Practical Guidelines for Business Success*. Harlow, England: Addison-Wesley, 1997.

[8] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture*. Chichester, UK: John Wiley and Sons, 1996.

[9] M. Klein and R. Kazman, "Attribute-Based Architectural Styles," Software Engineering Institute - Carnegie Mellon University, Pittsburgh, Technical Report CMU/SEI-99-TR-022, October 1999.

[10] H. A. Simon, *The Sciences of the Artificial*, Third ed. Cambridge, Massachusetts: The MIT Press, 1999.

[11] O. Preiss and A. Wegmann, "Stakeholder Discovery and Classification Based on Systems Science Principles," in *Proc. 2nd Asia-Pacific Conference on Quality Software (APAQS 2001)*, 2001, pp. 194-198.

[12] L. v. Bertalanffy, *General System Theory: Foundations, Development, Applications*. New York: George Braziller, 1969.

[13] J. G. Miller, *Living Systems*. Colorado: University Press of Colorado, 1995.

[14] P. Checkland and S. Holwell, *Information, Systems and Information Systems - making sense of the field*. Chichester, UK: John Wiley & Sons, 1998.

[15] A. Wegmann, "On the Systemic Enterprise Architecture Methodology (SEAM)," in *Proc. 5th International Conference on Enterprise Information Systems (ICEIS)*, 2003, pp. 483-490.

[16] J. Mylopulos, L. Chung, and E. Yu, "From Object-Oriented to Goal-Oriented Requirements Analysis," *Communications of the ACM,* vol. 42, pp. 31-37, 1999.

[17] B. Banathy. (2001, March). A Taste of Systemics. The Primer Project, A Special Integration Group of the International Society for the Systems Sciences (ISSS) [Online]. Available: http://www.isss.org/taste.html

[18] J. Bansiya and C. G. Davis, "A Hierarchical Model for Object-Oriented Design Quality Assessment," *IEEE Transactions on Software Engineering,* vol. 28, pp. 4-17, 2002.

[19] C. Swoyer. (2002, July). Properties. The Metaphysics Research Lab, Stanford University [Online]. Available: http://www.science.uva.nl/~seop/entries/properties/

[20] ISO/IEC, "Software engineering - Product quality - Part1: Quality model," ISO/IEC, International Standard 9106-1:2001(E)2001.

[21] I. Crnkovic, H. Schmidt, J. Stafford, and K. Wallnau, *Proceedings ICSE 4th international workshop on Component-Based Software Engineering*. Los Alamitos: IEEE Computer Society Press, 2001.

[22] O. Nierstrasz and D. Tsichritzis, *Object-Oriented Software Composition*. London: Prentice Hall, 1995.

[23] R. H. Campbell and A. N. Habermann, "The Specification of Process Synchronization by Path Expression," in *Proc.*

International Symposium on Operating Systems, 1973, pp. 89-102.

[24] O. Preiss, A. Frei, D. Gaidatzis, and A. Wegmann, "Comparison of XML Schemas for Asset Type Descriptions Compliant with RAS," ABB Switzerland - Corporate Research, Baden-Daettwil, White Paper, November 2002.

[25] Rational, "RAS - Reusable Asset Specification," Rational Software Corporation, Web document RAS 2001.09.04, Sep. 04, 2001.

[26] A. F. Ackerman, "Quality Grades for Software Component Source Code Packages," IEEE, International Standard, Informal Draft 0.1, 2001.