Establishing Software Development Process Control: Technical Objectives, Operational Requirements, and the Foundational Framework

James D. Arthur, Richard E. Nance, and Osman Balci

TR 92-40

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061

August 3, 1992

Establishing Software Development Process Control: Technical Objectives, Operational Requirements and the Foundational Framework

James D. Arthur, Richard E. Nance and Osman Balci
Virginia Tech
Blacksburg, VA 24061

Abstract

To produce and maintain product quality one must control the development and maintenance processes through the collection, examination and analysis of both process and product indicators. Process indicators provide measures that reflect the effectiveness of software development and maintenance activities. Product indicators provide measures that indicate the extent to which desirable, quality attributes are present (or absent) in the product (documentation and code). This paper proposes a foundational framework for establishing control over the software development process. Critical objectives stressed within this framework include: (a) the complementary integration of maintenance and development activities, (b) the identification and definition of a (semi-) automated data collection and analysis process which employs quality indicators that are definitively linked to the existence of process and product attributes, and (c) the formulation and use of control methods that are designed to work within the defined automated process and to provide decision support capabilities. The significance and necessity of these objectives are established through an examination of the Abstraction Refinement Model, the Objectives/Principles/Attributes Framework and the Software Quality Indicator concept.

1. PPOBLEM IDENTIFICATION AND SIGNIFICANCE

The critical nature and long life expectancy of today's complex software systems mandate the production of quality software products, and in particular, those that exhibit maintainability and reliability [PARD85]. Consistent with that mandate is a recognized need by the software engineering community to establish a controlled software development and maintenance process that (a) encourages the synthesis of quality products through feed-back control mechanisms and (b) provides risk assessment capabilities when pragmatic constraints force the consideration of sub-optimal alternatives. Current attempts to assess or predict product quality through the use of product metrics alone have met with significant criticism. Such efforts provide "after-the-fact" quality information, and have been described as being narrowly focused and providing measures that are often based on questionable metrics [KEAJ86]. Other research efforts have focused primarily on identifying the potential of a process to produce a quality product by questioning the existence of particular capabilities and characteristics of the development process. The implications being, that a process exhibiting such capabilities and characteristics will assure the production of a quality product. The Assessment of Contractor Capability developed at the Software Engineering Institute [HUMW87] is a manifestation of such research. While certainly a step in the right direction, determining the maturity level of a software development process does not necessarily guarantee the effective execution of that process leading to a quality product. We contend that to effectively *produce and maintain* a quality product one must control the development and maintenance processes through the collection and examination of

- process indicator measures that reflect the effectiveness of both software development and maintenance activities, and
- product indicator measures that convey the extent to which quality attributes are *observed* in the product.

In this paper we propose a foundational framework on which to build a controlled software development (and maintenance) process that instills product quality. The proposed framework is presented in Sections 2 - 4. Section 2 introduces the technical objectives that reflect appropriate needs and requirements. Section 3 provides a characterization of the software engineering framework, maintenance model and indicator measurement concept, that together, represent the definitional and developmental foundation. Section 4 discusses operational requirements derived from the technical objectives, and describes how those requirements are realized in an environment tailored for process control. Finally, a summary is provided in Section 5.

2. TECHNICAL OBJECTIVES

Consistent with our contention that <u>both</u> process and product indicators are crucial to establishing a controlled software development (and maintenance) process, we state three common-made observations, and identify the crucial

questions underlying them. Answers to these questions are found in the set of technical objectives reflecting the needs and requirements of a controlled process supporting *software evolution*, i.e. development *and* maintenance.

Observation 1: The maintenance activity represents a substantial portion of a product's life-cycle cost [HALD88]. To better understand and control attendant activities the maintenance process must be recognized as an integral part of the software development effort.

Question: *How do we establish a foundation for tracking process and product status throughout software development and maintenance activities?*

Technical Objectives:

- Recognize software as *evolving* from development, to acceptance, deployment and maintenance,
- Construct a model of the software development process that reflects pertinent standards and procedures, and
- Extend the model to depict the evolution of software and, specific requirements thereof.

Observation 2: Model abstractions support the characterization of "real world" objects and relationships among them. As such, they provide a fundamental basis for understanding and a blueprint for implementation.

Question: *Utilizing the software evolution model, how do we design and implement a metrics-based computer tracking system?*

Technical Objectives:

- Augment the model to include triggers and alerters to announce the need for human intervention and interaction, and
- Design a (semi-)automated system that embeds the evolution model and that maximizes Total Quality Management (TQM) objectives, while recognizing pragmatic constraints imposed by the existing development process.

Observation 3: To be useful, data collection and metric computations must be accompanied by powerful analytical techniques to support decision and control processes.

Question: *What statistical process control methods can be defined that complement the computer tracking system, emphasize the product quality, and are consistent with the TQM philosophy?*

Technical Objectives:

- Identify control methods for tailoring the embedded model to reflect organizational preferences,
- Define quality indicators for analyzing the implications of maintenance decisions and alternatives, and
- Within the software evolution model, identify fundamental relationships between indicator measures and product quality to facilitate reasoning about alternative scenarios and to promote understanding of quality and productivity implications at both technical and management levels.

The technical objectives stated above outline a *holistic* approach to establishing and maintaining control of the software development process. Integral parts of that approach are: (a) recognition that the development and maintenance processes are inextricably tied together, (b) the formulation of a model that succinctly captures the dependencies among and within phases of the development cycle, (c) the design of a (semi-)automated, metrics-based computer tracking system, and (d) the definition of process control methods that support technical and managerial decisions.

3. THE FOUNDATION FOR INTEGRATED DEVELOPMENT

The crafting of a controlled software development process requires that the execution of all technical objectives be guided by an integrated development approach which promotes understanding and reasoning at various levels of abstraction. In particular, this crafting activity requires a well-defined set of capabilities (or requirements), and a foundation on which to realize those capabilities. Elements crucial to such a foundation are: (a) a framework that justifiably links prescribed activities to the achievement of desirable software engineering qualities and that supports a definitive process for evaluating alternatives among these activities, (b) a model that describes the integral relationship between software development and maintenance, and (c) a measurement approach that directly links measurable properties of the process and product to non-measurable concepts. This section describes each of these essential elements and discusses their particular contributions toward establishing a controlled software development process.

3.1 A Characterization of the Software Development Process

In the late 1800s Lord Kelvin recognized the crucial role of metrics in the management process. Stated in paraphrased form, his contention is that "you cannot manage what you cannot measure." Today, metric-based analysis appears to be the most promising approach to controlling the software development process. Unfortunately, many metrics currently in use are non-intuitive, non-instructive, and lack that fundamental basis for understanding the implications of one measurement value as compared to another. While we too propose the definition and use of metrics, we do so within a guiding framework that embraces intuitive measures and provides a basis for reasoning

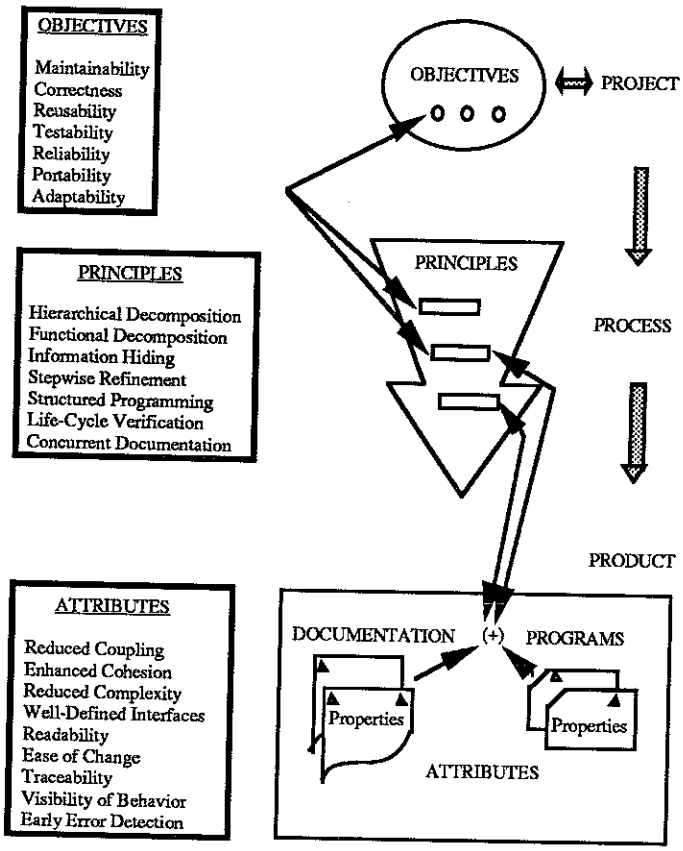

Figure 1
Illustration of the Relationship Among Objectives, Principles, Attributes
in the Software Development Proces

about the implication and ramifications of those measures grounded in software engineering concepts. That guiding framework, called the "Objectives/Principles/Attributes Framework," is discussed below.

3.1.1 The Objectives/Principles/Attributes Framework

The Objectives/Principles/Attributes (OPA) framework [ARTJ90] characterizes the *raision d'etre* for software engineering; that is, it embodies the rationale and justification for software engineering. As illustrated in Figure 1, the framework enunciates definitive linkages among project level objectives, software engineering principles, and desirable product attributes. In particular, it advances the following rationale for software development:

- a set of objectives can be defined that correspond to project level goals and objectives,
- achieving those objectives requires adherence to certain principles that characterize the process by which the product is developed, and

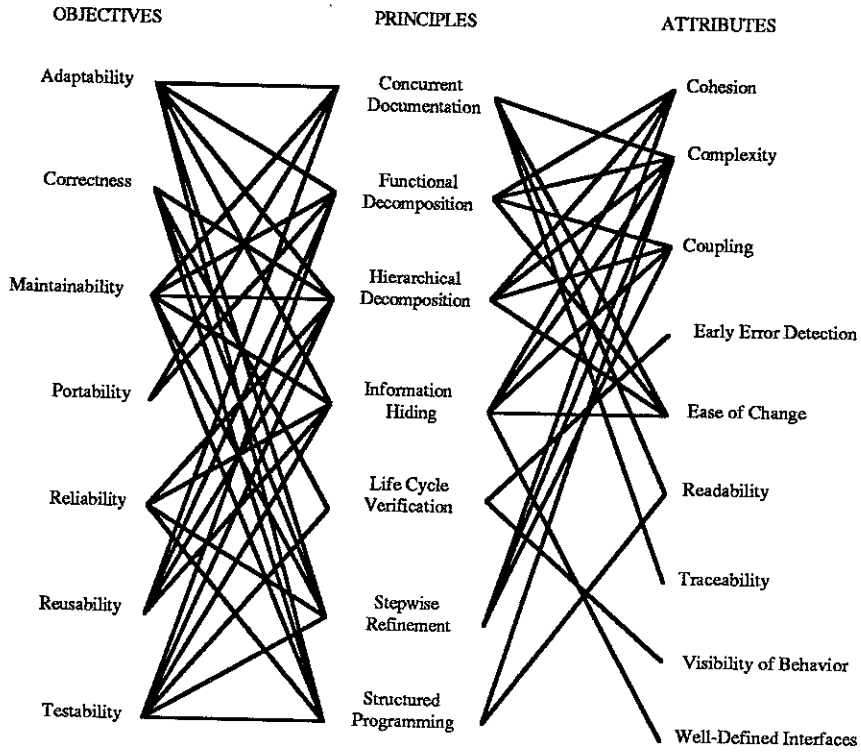

Figure 2
Linkages Among the Objectives, Principles and Attributes

- adherence to a process governed by those principles should result in a product that possesses attributes considered to be desirable and beneficial.

Underlying this rationale is a natural set of relations, depicted in Figure 2, that link individual objectives to one or more principles, and each principle to one or more attributes. For example, to achieve maintainability one might employ the principle of information hiding in the development process. In turn, employing information hiding will result in a product that exhibits a well-defined interface.

A natural question at this point is: *How does one determine if, and to what extent, a product possesses desirable attributes?* The answer lies in the observation of product properties, i.e. observable characteristics of the product. For example, the use of global variables indicates that a module interface is not well-defined [DUNH80, p. 149]. The number of global variables used relative to preferable forms of communications, e.g. parameter passing, indicates the extent to which the interface is ill-defined.

In "bottom-line" parlance: (1) the achievement of software engineering objectives is directly linked to the use of specific principles, (2) as a consequence of using these principles desirable attributes are induced in the product, (3)

by observing product properties to determine the extent to which desirable attributes exist in the product, one can ascertain the extent to which particular principles are governing the development process, and in turn, the extent to which stated software engineering objectives are achieved. To date, we have defined and substantiated (through published results of independent researchers) 33 linkages among the seven prominent software engineering objectives and nine principles, 24 linkages among principles and attributes, and 84 property/attribute pairs [ARTJ87].

3.1.2 Contributions of the OPA Framework to Controlling the Software Development Process

Controlling the software development process necessitates a systematic approach to assessing product and process conformance to acceptance standards. How, then, does the OPA framework support such an approach?

(1) Through its property/attribute pairs and linkages relating attributes to principles and principles to objectives, the OPA framework supports a well-defined, systematic approach to examining product and process quality. In particular, observable product properties, directly linked to attributes, provide a basis for evaluating the product. Similarly, observable characteristics and trends of the process support process assessment, e.g. requirements change following the software specification review indicates a lack of early error detection [YUTJ88, p. 1268], the creation and use of software development folders promote traceability and visibility of behavior [MCMJ87, p. 70].

(2) The OPA approach provides a rigorous framework for: (a) relating acceptance criteria based on attributes to software engineering principles and objectives, and (b) defining acceptance levels based on measures reflecting the achievement of objectives, adherence to principles and realization of attributes. That is, the extent to which a product or process exhibits desirable attributes indicates the use of particular software engineering principles and the likelihood of achieving desirable software engineering objectives. Assessing process and product conformance to acceptance standards based on attribute, principle and objective measures provides a well-defined feedback mechanism for controlling the software development process.

(3) Finally, the OPA characterization of the software development process provides a basis for interpreting process and product quality measures, and thereby, permits effective process control. For example, if one observes a value indicating the achievement of a software engineering objective, and that value is not consistent with expectations, then contributing principles are examined (based on the defined linkages among objectives and principles) for anomalous values. Similarly, the linkages among principles and attributes point to candidate attributes to be examined for suspect values. Finally the attribute/property relations enable the identification of the most prominent process or product characteristic(s) influencing the original objective value. The identification of an anomalous attribute/property pair indicates the misuse (or non-use) of a critical software engineering principle. The points where this principle is most apparent in the process become the prime

candidates for attention. With appropriate reporting one can also determine the offending product component. Detecting anomalous process characteristics follows a similar progression.

3.2 Process and Product Examination Through Direct, Yet Intuitive, Measures

Economic and social indicators are based on the thesis that intangible, qualitative conditions can be indirectly assessed by measurable quantitative characteristics [CARE79, pp. 9-11]. Controlling the software evolution process mandates that we too must be able to measure the unmeasurable, e.g. traceability, the use of concurrent documentation, and the achievement of portability. In support of such measures, a concept analogous to the economic and social indicators is advanced in the OPA framework: Software Quality Indicators. Software Quality Indicators are embodied in the OPA framework through attribute/property relationships. For example, an intangible attribute of the development process, like early error detection, can be indirectly assessed through measurable properties, like the changing of requirements after the software specification review. For clarification purposes we note that our use of the term "Software" in "Software Quality Indicators" is not intended to be restrictive, but applicable to both process and product quality indicators.

3.2.1 A Characterization of Software Quality Indicators

A Software Quality Indicator (SQI) is a variable whose value can be determined through direct analysis of product or process characteristics, and whose evidential relationship to one or more attributes is undeniable [ARTJ87, p. 25]. Crucial in this working definition is that

- the value is directly measurable through the analysis of the software development process or products of that process, e.g. programs and documentation, and
- SQIs are always attribute/property pairs
  denoting undeniable relationships, and
  indicative of the presence or absence of one or more attributes.

Consider, for example, an SQI based on code analysis: coupling through the use of structured data types (CP/SDT). The property in this SQI is the use of structured data types, and the attribute is coupling. One can argue that the use of a structured data type as a parameter argument has a detrimental impact on module coupling. That is, structured data types allow the consolidation of related data items. When passed as a parameter, however, rarely is every data item in the structure accessed by the called module. Consequently, these extraneous items unnecessarily increase the coupling between the calling and called modules. [TROD81, p. 115]. A candidate measure is the ratio of the number of structured data type used as parameters relative to the total number of parameters. Note that (a) the value is

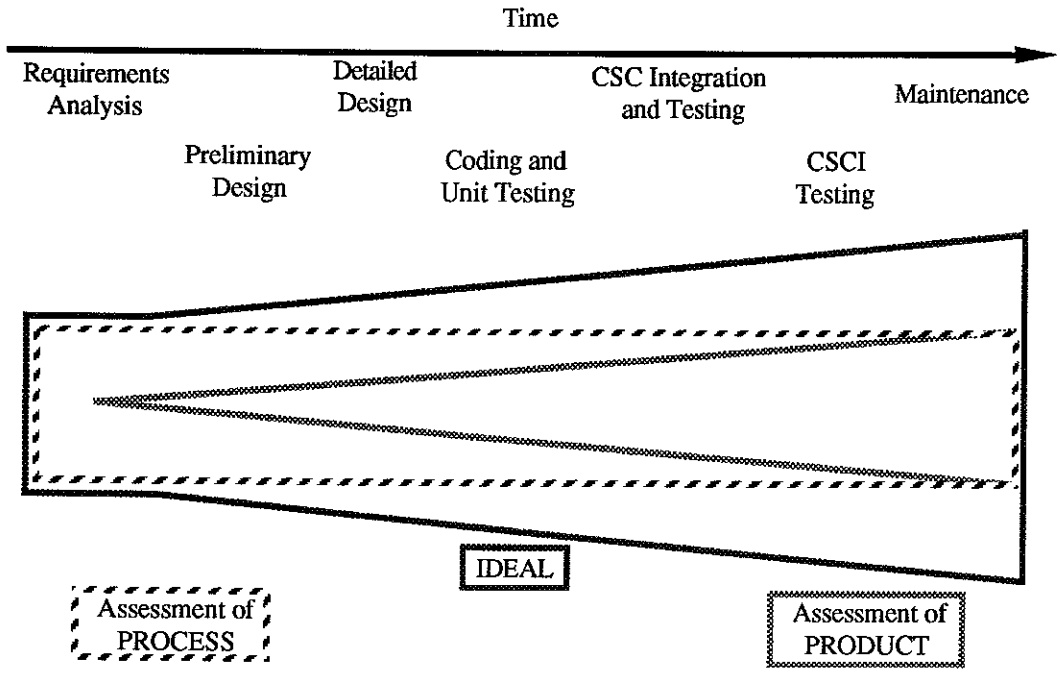

Figure 3
Exploiting both Process and Product Indicators

directly measurable, (b) the SQI is an attribute/property pair, (c) the relationship described between the use of structured data types and coupling is undeniable (and intuitive), and (d) the stated SQI can indicate the presence (or absence) of coupling between two modules.

As described above, because the use of structured data types can impose unnecessary coupling, module maintainability is reduced. Paradoxically, one might also argue that the use of structured data types have a positive impact on module maintainability. In particular, because structured data types allow one to group related objects together, they promote cohesion (which promotes a more maintainable module [CONS86, p. 108]). *What then allows us to distinguish between a beneficial use of a property and a detrimental one?* The answer is the association with an attribute; that is, the attribute/property pair denotes the SQI, and not the property alone. This observation is crucial because it points to the fact that indicators can be used to confirm the beneficial contribution of a property in one instance and the detrimental effect in another. Ideally, we would like to have many confirming and contrasting SQIs to capture a holistic view, and to convey how well we are (or are not) carrying out the development process. Coupled with the linkages within the OPA framework (through the attributes) we can reason, from a software engineering standpoint, (a) *why* particular values are being reported, and (b) *how* to correct deficiencies to effect improvement.

In addition to the characteristics outlined and implied by our working definition, SQIs should also be

- simple, understandable and easily related to attribute(s),
- targeted at process activities, design information and implementation products, and
- as objective as possible.

3.2.2 Controlling the Software Development Process Through Software Quality Indicators

As characterized above, Software Quality Indicators are indispensable in monitoring and controlling the software development process. In particular, they provide the feedback information (a) to judge the effectiveness of the many process-related activities, and (b) to base decisions for effecting process improvement. Because software evolution begins with requirement specification activities and continues throughout the life of the product (including attendant maintenance activities), SQIs must embrace both process and product measures and minimally must admit to semi-automatic computation.

(1) As illustrated in Figure 3, we propose the use of SQIs throughout the product software life cycle. Initial SQI measures must reflect process-oriented characteristics because little, if any, product is available. As development continues and products become more readily available, SQI measures should expand correspondingly to reflect product-oriented qualities. Preliminary work in the SQI domain suggests that process, documentation and code indicators are needed [ARTJ91, p. 5].

- Process indicators exploit byproducts of process activities and trend data. They fall into one of three categories: phase independent, phase dependent and phase spanning. Development instability as related to early error detection, requirements volatility as it affects traceability and cohesion, and the creation of software development folders relative to traceability are representative SQIs of each category, respectively. Ten process indicators are currently being examined for their effectiveness in predicting product quality

- Documentation quality indicators attest to document accuracy, completeness and usability. For example, establishing factual consistency within and among documents is desirable when assessing accuracy. Currently we are investigating 36 candidate document quality indicators. One aspect of this investigation is to judge the compatibility and utility of these indicators within a development process conforming to DoD-STD-2167A.

- Code indicators relate software engineering attributes to code properties. For example, the use of packages in Ada program allows us to assess functional cohesion. To date, we have identified over 100 code

indicators, 66 of which have been automated. Our investigation of code indicators has focused on Ada, CMS-2 and Pascal.

(2) Experience has shown that process-related assessment must, to a large extent, be automated. Because our proposed approach to controlling the software development process utilizes many contrasting and confirming SQIs, the data items required to compute them are simple and easily obtained. Subsequently, automated data collection is facilitated, as is the metric computation associated with each SQI. As mentioned above, we currently have a prototype Ada analyzer and report generator that employs 66 code indicators. A feasibility study addressing the automation of document quality indicators is underway; a portion of that study has included the development of a prototype document analyzer that extracts the necessary data items for a subset of the document quality indicators.

3.3 Integrating the Maintenance Activity into the Software Development Life Cycle

Descriptions of software evolution are referred to as "life-cycle" models. These models try to represent the process by which software systems evolve. Included are the activities that recur throughout the life of a system (hence, the term "cycle"). Technically, these models should make no distinction between development and maintenance; rather; they should focus instead on the abstract activities that occur in both. Unfortunately, the conventional train of thought has been to "completely" specify development phases and activities, and then address maintenance as an "add-on" (if at all). The deemphasis on maintenance is representative of a lack of understanding of what maintenance means. This is especially disconcerting when even the most conservative estimates project that software maintenance contributes 50%-70% of the life-cycle costs [HALD88, p. 236].

Clearly, our contention that maintenance should be considered as an integral part of the life-cycle process and share the same priority as other life-cycle phases is supported. Increased understanding of the maintenance activity, better risk assessment capabilities, and enhanced quality control process are only some of the potential benefits. The Abstraction Refinement Model [NANR89] discussed below characterizes the inherent linkage between maintenance and development, and provides a basis for reasoning about the benefits of merging the two "activities" within a model of software evolution.

3.3.1 A Characterization of the Abstraction Refinement Model

The Abstraction Refinement Model (ARM) characterizes the inherent dependency of the maintenance function on the software development activities through the required documentation produced in the development process. For simplicity, consider software development originating from a set of software specification requirements. One can view the development process as a sequence of steps along some path leading to the realization of program **P**. This

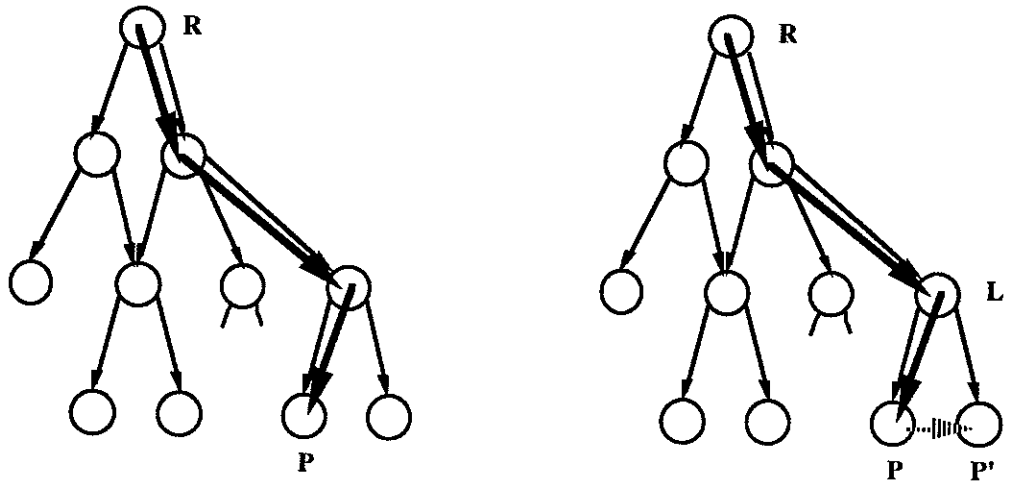

Figure 4
Resolution of Program Abstraction Through Development Activities

path is marked by transformations, taking a prior abstract specification to a less abstract (more concrete) specification. At each step, however, alternative resolutions are possible. Many of the alternatives are not consistent with the requirements and are not taken, while other alternatives are acceptable and simply imply different paths (approaches) leading to the same realization of program **P**.

Figure 4a illustrates the stepwise progression towards **P** where successive nodes along a path represent increasingly refined abstractions of the requirements and the arcs represent transitions to accomplish refinement. That path, starting at node **R**, is constructed through design decisions that effect refinements of higher level specifications to realize specifications at a lower level. By capturing (a) design decisions in the documentation of the development process and (b) the refinement changes induced by the design decision, reusability of the refined component is accommodated. This recorded set of decisions and changes is often referred to as the system context for a node (or refined program component) [NEIJ84, p. 565]; each refined representation has its own unique system context. As outlined next, the system context plays an important role in software maintenance.

Software maintenance can be characterized as a transformation of program **P** to **P'**. Figure 4b illustrates the desire to move from the concrete representation **P** to **P'**. All too often maintenance programmers invoke the "blind transformation" approach to maintenance. That is, the maintenance programmer attempts to realize **P'** from actions on **P**, neglecting any predecessors of **P** (like **L**) whose system context is the same for both **P** and **P'**. An example of such an attempt is the correction of a program using only the source code and internal documentation.

If the system context has been maintained during the development process (and subsequent maintenance activities) then a more appealing approach is to apply reverse engineering procedures to move from **P** to **L**, and the forward

engineer to **P'**. Alternatively, one might also start with **L** to derive **P'**, recognizing that **L** is the least common abstraction of both **P** and **P'**.

We recognize that pragmatics argue against the recording of all decisions and attendant changes. Nonetheless, the Abstract Refinement Model does allow one to examine a development process that includes maintenance and observe the potential benefits that can be derived from recording decisions and changes. From a contrary perspective, the ARM reveals the detrimental impact if the contextual needs for maintenance are ignored during development. Moreover, the ARM underscores the importance of *maintaining* the system context rather than the implementation alone.

3.3.2 The Necessity of Linking and Controlling Both Development and Maintenance Activities

Through the Abstraction Refinement Model we can view an ideal software development cycle that integrates both maintenance and conventional development activities. The ideal integration may not be practical, but at least a goal is set forth, and the ramifications of the gap that separates the existing development life-cycle from the ideal one are conveyed. We do propose, however, to narrow that gap by pursuing a realistic life-cycle model that reflects integration tempered by pragmatic constraints. As outlined below, the ARM is used as a guide for the development of additional control mechanisms.

(1) The ARM emphasizes the importance of tracking and recording decisions and changes. Consequently, development and maintenance activities that effect transformations should be monitored to insure adequate and accurate recording of decisions.

(2) The characterization documentation from a system context perspective reveals the critical need to keep such documentation current. Recognizing constraints imposed by storage and update costs, the ARM suggests a risk assessment model that relates maintenance forms to the probabilistic availability of requisite documentation commensurate with each form.

(3) Finally, because the ARM emphasizes the importance of adequate documentation, the necessity and utility of document quality indicators in controlling the development process is illustrated.

4. REQUIREMENTS FOR AN INTEGRATED PROCESS CONTROL ENVIRONMENT

Guided by the technical objectives stated in Section 2, the use of software quality indicators within the Objectives/Principles/Attributes framework, and the employment of the Abstraction Refinement Model to link

software development and maintenance represent a holistic approach to establishing an integrated environment tailored to process control. That control can only be realized, however, through

- the formulation of a model that (a) elevates maintenance to its proper place in the development life-cycle, and (b) advances a framework embodied in fundamental software engineering concepts,

- the design of a (semi-)automated computer tracking system that exploits software quality indicators to provide measures that are intuitively linked to desirable software engineering attributes, and

- the development of process control methods and techniques that emphasize decision support, risk assessment and reasoning about a software evolution process based on fundamental principles.

These three tasks reflect our perceptions of a necessary and sufficient approach to establishing a controlled software development process through integrated instrumentation and feedback analysis. The three tasks cannot, however, be performed in isolation. The execution of each task must reflect the needs and requirements of the others. The remainder of this section addresses each task individually, relates each task to the SQI, OPA and ARM concepts, and outlines requirements for their realization.

4.1 A Unified Model of Software Development and Maintenance

Integral to software evolution is a foundation that supports the tracking of process and product status throughout development and maintenance phases. That foundation should permit the formulation of abstractions to hide unnecessary details and thereby facilitate the examination of development and maintenance alternatives in a controlled, manageable manner, providing a blueprint for implementation.

Establishing the foundation requires that development and maintenance phases first be integrated into a single, cohesive model that promotes the recognition of maintenance requirements during development.

- The proposed model should recognize the specific needs of both development and maintenance activities by linking them through a common documentation set that records design decisions and alternatives. The Abstraction Refinement Model (ARM) described in [NANR89] and outlined in Section 3.3 specifically addresses the necessity of and benefits derived from such an approach.

- The proposed model should support a characterization of the four maintenance forms relative to development documentation needs and suggest risk assessment possibilities.

- The model should provide a basis for defining a maintenance process that both enunciates and employs reverse and forward engineering principles.

In general, the model provides the framework for recognizing, defining and evolving a methodological approach that permits a realization of the symbiotic relationship between development and maintenance activities throughout the entire software life-cycle.

4.2 An Indicator-Based (Semi-)Automated Tracking System

Previous experience has shown the absolute necessity of a (semi-)automated tracking, data collection and report generation system [NANR85]. The Objectives/Principles/Attributes (OPA) characterization of software development [ARTJ90] and the Software Quality Indicator concept [ARTJ87] ideally support the definition of and justification for requisite measurement approaches and corresponding metrics. The model described above facilitates the examination of both the development methodology and process to determine what metric data is needed, where such data is available, and how to extract it..

- As a preliminary step toward implementing the computer-based tracking system, the development/maintenance model should be augmented to include triggers and annunciators to signal the need for human assistance, interaction, and possible intervention.

- Model abstractions representing control resources should be mapped onto the development process to permit the identification of critical points where constant monitoring is needed, e.g. where unchecked deviations from established threshold values imply undesirable consequences.

The design of a (semi-)automated system that embeds the development/maintenance model is the next logical step. The system cannot, however, be developed in a vacuum.

- The design process should recognize the organization's approach to TQM for software-intensive projects, while anticipating, but not being "blinded" by, pragmatic constraints imposed by the existing development/maintenance process.

- Based on the Objectives/Principles/Attributes framework for software development, the appropriate set of Software Quality Indicators (SQIs) should be identified and characterized. In particular, they must definitively measure (a) the extent to which software engineering attributes are present (or absent) in the process and product, (b) the use of selected software engineering principles in the development process, and (c) the achievement of desired software engineering objectives.

- SQI definitions should employ (a) process measures reflecting an examination of process activities characteristics and trends, and (b) product measures reflecting observable code and documentation properties.

- Measurement approaches should be justifiably linked to defining SQIs and the corresponding concepts they purport to measure.

- Metrics supporting each measure should employ data items that confirm or refute the existence of desirable process and product attributes.

- Control points should be identified within the development/maintenance model where data collection and SQI measurement is critical to achieving a holistic process/product assessment strategy. (Data collection and SQI measures can be embedded at corresponding points in the software development process.)

Concurrent with the design phase must be the explicit recognition of Software Quality Assurance (SQA) and Configuration Management (CM) roles in an environment that provides software development process control.

- An additional SQA activity should be added to utilize automatic feedback from product analysis to monitor the process, leading to an established (or at least encouraged) CM activity that is tightly controlled.

4.3 Statistical Process Control Reflecting Objectives of a TQM Organization

Effective statistical process control within a software development environment is dependent on the identification of control methods that complement the computer tracking system. We propose innovative control methods as an integral part of the automated system described earlier.

- To augment decision support capabilities of management and technical personnel, control methods should be defined that clearly enunciate implications associated with decision alternatives.

- Control methods should be defined to accommodate changes in the software development process and in organizational priorities.

- Recognizing the interdependencies among development and maintenance activities and the necessity of an integrated control environment, control methods must be defined which fit naturally into the software evolution model embedded in the software development process.

- For productivity and human engineering purposes, control methods should incorporate terminology preferences and reflect organizational policies.

- Control methods associated with maintenance activities should be designed to measure the quality level of software components affected during maintenance. They should provide quantitative evidence indicating whether the level of quality has been retained or improved, and if not, at what cost. (Incidentally, this "metrics-driven" approach provides management with a tool for evaluating the effectiveness of the maintenance organization.)

As evidenced in the above discussions, we argue for and advocate (1) a comprehensive model of software and maintenance activities and (2) a computer tracking system and control methods based on the OPA framework, and on the use of SQIs. These choices are intentional because the approach results in indicator measures that can be directly linked to product quality through intuitive arguments. These characteristics encourage management and technical personnel to question the "whys" of conventional wisdom and the "what ifs" of proposed changes.

5. SUMMARY

Achievement and retention of product quality requires controlling the development and maintenance processes through the collection, examination and analysis of both process and product indicators. Process indicators provide measures that reflect the effectiveness of software development and maintenance activities. Product indicators provide measures that indicate the extent to which desirable, quality attributes are present (or absent) in the resulting product (documentation and code).

Based on previous experience, we propose a foundational framework that can be used as a springboard for establishing software development process control and provides a blueprint by which organizations can establish a controlled software development process. In particular we advocate

- the formulation of a software development model that recognizes the importance of maintenance activities in development life-cycle phases through a synergistic integration of development and maintenance activities,
- the design of a (semi-)automated system that provides decision support capabilities through the use definitive process and product indicators, and
- a characteristic definition of control methods that provide the basis for establishing and maintaining process control through quality feedback analysis and risk assessment.

The synthesis of a software development and maintenance environment that exhibits desirable control characteristics, however, must be built upon a foundation reflecting

- a systematic approach to assessing product and process conformance to acceptance standards: effectively, an Objectives/Principles/Attributes framework,
- product and process examination through direct, yet intuitive measures: like those represented by Software Quality Indicators, and finally,
- the inherent dependencies of maintenance activities on software development activities through documentation produced in the development process: the precise quality represented by and embodied in the Abstraction Refinement Model.

REFERENCES

[ARTJ87] Arthur, J.D. and R.E. Nance, "Developing an Automated Procedure for Evaluating Software Development Methodologies and Associated Products," Technical Report SRC-87-007, Systems Research Center, Virginia Tech, Blacksburg VA, April 1987.

[ARTJ90] Arthur, J.D. and R.E. Nance, "A Framework for Assessing the Adequacy and Effectiveness of Software Development Methodologies," *Proceedings of the Fifteenth Annual Software Engineering Workshop*, Process Improvement Session, Greenbelt MD, December 1990.

[ARTJ91] Arthur, J.D., Nance, R.E., Bundy, G.N., Dorsey, E.V. and J. Henry, "Software Quality measurement: Validation of a Foundational Approach," Technical Report SRC-91-002, Systems Research Center, Virginia Tech, Blacksburg VA, 1991.

[CARE79] Carmines, E.G. and R.A. Zeller, *Reliability and Validity Assessment*, Quantitative Applications in the Social Sciences, J.L. Sullivan (Ed.), Sage Publications, Beverly Hills CA, 1979.

[CONS86] Conte, S.D., Dunsmore, H.E. and V.Y. Shen, *Software Engineering Metrics and Models*, Benjamin/Cummings Publishing Co., CA, 1986.

[DUNH80] Dunsmore, H.E. and J.D. Gannon, "Analysis of the Effects of Programming Factors on Programming Effort," *Journal of Systems and Software*, Vol. 1, No. 2, February 1980, pp. 141-153.

[HALD88] Hale, D.R. and D.A. Haworth, "Software Maintenance: A Profile of Past Empirical Research," *IEEE Conference on Software Maintenance*, Scottsdale AZ, October 1988, pp. 236-240.

[HUMW87] Humphrey, W.S. and W.L. Sweet, "A Method for Assessing the Software Engineering Capability of Contractors," CMU/SEI-87-TR-23, Software Engineering Institute, Carnegie Mellon University, Pittsburgh PA, September 1987.

[KEAJ86] Kearney, J.K., R.L. Sedlmeyer, W.B. Thompson, M.A. Grey and M.A. Adler, "Software Complexity Measurement," *Communications of the ACM*, Vol. 29, No. 11, November 1986, pp. 1044-1050.

[MCMJ87] McManus, J.I., "The Heart of SQA Management: Negotiation, Compliance, Regression," Handbook of Software Quality Assurance, G.G. Schulmeyer and J.I. McManus (Eds.), Van Nostrand & Reinhold, NY, 1987.

[NANR85] Nance, R.E., Arthur, J.D. and A.V. Dandekar, "Evaluation of Software Development Methodologies: Final Report of the Immediate Software Development Issues Project," Systems Research Center, Virginia Tech, Blacksburg VA, December 1985.

[NANR89] Nance, R.E., Keller, B.J. and D. Boldery, "Document Production Under Next Generation Technologies," Technical Report SRC-89-001, Systems Research Center, Virginia Tech, Blacksburg VA, February 1989.

[NEIJ84] Neighbors, J.M., "The DRACO Approach to Constructing Software from Reusable Components," IEEE Transactions on Software Engineering, SE-10, No. 5, September 1984, pp. 564-574.

[PARD85] Parnas, D.L., "Software Aspects of Strategic Defense Systems," *Communications of the ACM*, Vol. 28, No. 12, December 1985, pp. 1326-1335.

[TROD84] Troy, D.A. and S.H. Zweben, "Measuring the Quality of Structured Design," Vol. 4, No. 2, June 1981, pp. 113-120.

[YUTJ88] Yu, T., Shen, V.Y. and H.E. Dunsmore, "An Analysis of Several Software Defect Models," *IEEE Transactions on Software Engineering*, Vol. 14, No. 9, September 1988, pp. 1261-1270.