Improving the Quality of Software Quality Determination Processes*

Leon J. Osterweil
Department of Computer Science
University of Massachusetts
Amherst, MA 01003
USA

Abstract

This paper suggests a systematic, orderly, process-based approach to stating software quality objectives and knowing if and when they have been achieved. We suggest that quality in software is a complex, multifaceted array of characteristics, and that it is important to establish specific objectives along various software quality dimensions as requirements for software quality assurance determination processes. We propose that process technology be used to design, code, execute, evaluate, and migrate processes that are demonstrably effective in achieving required software product quality objectives. Recently there have been numerous highly visible efforts to codify the assessment of software processes, and to use assessment results to improve them. In this paper we argue that these efforts function as testplans for software processes. We borrow some of the notions proposed in these efforts, and indicate how they can be used to construct a discipline of measuring and evaluating how well processes can be expected to deliver specific knowledge about software product qualities. We look towards the gradual, but eventual, establishment of an orderly discipline of software quality demonstration process development that should ultimately support a marketplace in which definitive knowledge about the nature of software products can be bought and sold.

Keywords:
software process, software quality, process improvement, integrated software testing and analysis

1 INTRODUCTION.

As the use of computers becomes more pervasive in our civilization and society, it becomes increasingly important to be sure that the software used to direct and control them is of high quality. Computer systems now support virtually all infrastructural areas of society. They are essential in banking. They support key medical functions. They facilitate educational processes. They are used to help design roads and buildings and to coordinate their construction. They are the backbone of our communications systems. They are essential to such national security functions as defense and intelligence. Computer systems are now increasingly integral to the practice of science, to many forms of the arts, and to the humanities and social sciences. Indeed it is increasingly clear that computer systems are themselves critical societal infrastructure.

All of this underscores the importance of being sure that computer systems do their jobs satisfactorally. Unfortunately this is evidently often not the case. Computer systems routinely fail in use. Often these failures are merely annoying, as when a bill for $0.00 arrives. Sometimes the results are amusing, as when a form letter is comically misaddressed. But computer failures can be serious, as when a New York bank

*This work was supported in part by the Air Force Materiel Command, Rome Laboratory, and the Advanced Research Projects Agency under Contract F30602-94-C-0137

had to borrow several billions of dollars because its computer failed to reconcile its books at the end of a business day. Failures can cause serious injury or death, as when a computer system administered lethal doses of radiation to patients. It is easy to see that more serious failures, for example in defense systems, can potentially cause large scale death and destruction. In the past, most failures have been attributable to software, rather than hardware. Thus it is clear that assuring that computer software is of high quality, and will not fail, is an issue of very great importance.

Assessing the quality of software and assuring that it will not fail is difficult and complex. Much of the complexity is because there are many dimensions of quality, and software can fail in many different ways. Functional correctness is the most obviously important quality. Certainly we rely upon software to compute the right answers. But other quality attributes are also important, and may sometimes be of more important than functional characteristics. For example speed is critically important, especially in interactions with human users, and in real-time monitoring. Robustness is also usually quite important, but is often overlooked. Computer systems must react responsibly when they receive incorrect and unexpected inputs. They must avoid certain behaviors and must not fail catastrophically. Other quality characteristics that software is generally expected to demonstrate, are reliability, comprehensibility, safety, user-friendliness, and modifiability. The list could go on. It is also worth noting that these characteristics may conflict with each other. Thus, execution speed is often sacrificed to achieve reliability and robustness (for example).

It is a real challenge to determine the extent to which a piece of software demonstrates a particular quality. For example, it is very hard to determine if a piece of software can ever possibly fail to compute correct functional values. Even a modest-sized piece of software may compute dozens of functions, and perhaps different sets of functions for different inputs. The inputs to any function may come from a potentially infinite space of possibilities. Testing merely samples from that space. How does one determine when the sampling has been sufficiently representative? Similar problems arise in testing software speed. Does one determine worst case performance? or average case? If one is interested in the average case, then over what sample does one average? Assessing the readability, modifiability, and user-friendliness of software poses serious challenges of still different kinds.

2 APPROACHES TO SOFTWARE QUALITY.

2.1 Testing.

Post-development testing is the traditional way to determine and assure quality. There are obvious advantages to exercising the completed software with specimen input data. Program behavior can be observed in the actual deployment environment to enable study of interactions with runtime support and operating systems. Program instrumentation can be used to support arbitrarily intensive scrutiny of the program's execution. Thus, failures can be identified and studied in minute detail, and evolutions of values of variables can be studied. Flow of control details can be observed, recorded, and analyzed.

But dynamic testing has serious drawbacks as well. Instrumentation inevitably perturbs the program's execution. Thus, the instrumentation may cause new failures, or may mask and distort previous failure phenomena. A more fundamental problem with dynamic testing is that it can demonstrate the presence of faults leading to failures, but it is generally not able to demonstrate their absence. Early in testing, failures are expected, and testing leads to fault correction. But later it is increasingly hoped that failures will not be detected, and faults are no longer present. As determining the absence of faults becomes the key goal, dynamic testing becomes increasingly inappropriate.

When dynamic testing is used to show the absence of faults it is essential to select test data sets that thoroughly exercise the program. An extensive literature documents many approaches to selecting test data [How87]. Some emphasize systematic sampling of the program's input data space (black box,

or requirements-based, testing) (eg. see [RAO92]). These approaches tend to emphasize modes in which actual users tend to use the program. Their weakness is that they pay less attention to less frequently-used modes, such as failure recovery, that may still be quite important. They also fail to concentrate on implementation structures that may be error-prone. Thus a second major test data selection approach is so-called white box testing, in which test cases are chosen to assure that implementation structures are thoroughly exercised [How78, RT86]. Some combination of these approaches seems indicated.

2.2 Static Analysis.

Static analysis complements dynamic testing in its ability to show the absence of certain classes of faults. It has the additional advantage of not requiring execution of the program, and hence the selection of test data sets. Static analyzers work on models of the program and its possible executions. The most familiar static analyzers are incorporated into compilers. Compiler syntax analyzers build and examine parse trees and determine the presence or absence of syntactic faults. More ambitious compilers, and similar standalone analysis tools, determine the presence or absence of certain semantic faults (eg. the PFORT Fortran Analyzer [Ryd74], and the Lint C analyzer).

More powerful static analyzers build increasingly sophisticated program models, and employ increasingly powerful mathematics to demonstrate the absence of wider classes of faults. Thus dataflow analyzers [OF76, DC94] construct annotated flowgraphs to represent program execution behaviors, and then compare them to regular expressions that describe sequences of events that represent desired behaviors. Experience suggests that these models help analysts study such qualities as safety and robustness, but are less helpful in studying functional properties. Formal verification [AGB+77, Bjo87] uses such formalisms as predicate calculus to represent program functional behaviors. Inference techniques such as symbolic execution [Kin75, Cla76] then develop models of program functionality that are then compared to desired functional behavior specifications to determine the presence or absence of functional faults.

While these analyses can demonstrate the absence of some classes of faults, it is important to note that they cannot determine the absence of all kinds of faults. it is also important to note that these analyzers check models of execution behavior, rather than actual execution behavior, and may therefore yield inaccurate or incomplete results.

2.3 Development Approaches.

While testing and analysis of finished code remains the most popular approach to assuring software quality, there is considerable sentiment that it is more effective to build quality into software as it is being developed. This philosophy is consistent with practices in industries such as automobile manufacturing. Building quality into software entails embedding quality assurance steps and practices into the development process. There is now a rough consensus about the overall strategy for doing this. For example, it is now agreed that finished software should contain a specification of its requirements, various design representations, testcases and their corresponding test results, code, and documentation. It is also agreed that different of these artifacts need more attention during different development stages. Thus, building quality into the product entails assuring that each new artifact is properly developed and integrated with the others. Thus, for example, as design specifications are built, they should be shown to be responsive to stated requirements; as code is built it should be shown to correctly implement the corresponding designs; after coding is complete, a testing process should demonstrate that test case execution results are consistent with requirements. This seems to be a useful general prescription for developing quality software, but it lack the details of precisely how and when to do all of this.

2.4 Integration of These Approaches.

While it is reasonable to build quality into software as it is being developed, it also seems essential to evaluate the software after development to be sure that the desired quality has been achieved. It seems clear that some sort of combination of testing the finished code, and analyzing models of it for faults is needed. Many also believe that this sort of combination is also what is needed to assess quality on an ongoing basis during development. As indicated above, it seems agreed that integrating these approaches is desirable, but there is no agreement about the details of how to do this.

3 THE ROLE OF SOFTWARE PROCESS.

Software process work supports building quality into software by focusing on how software is built and assessed. The software process community regards software as a complex, intricately interconnected information aggregate that has been built by executing a process that should be orderly and systematic. A software product can be considered to be of high quality only when its constituent artifacts have been shown to be consistent with each other, and within themselves, in a variety of ways. Evidence of these forms of consistency is captured by other software artifacts, generally reports from such quality assurance activities as analysis and testing. Thus, high quality software products are necessarily large and complex, and the processes by which these complex structures are assembled and assessed must be appropriately complex as well. Unfortunately software products are generally developed using informal ad hoc processes that usually reside in the heads of those who execute and supervise them. As a consequence, software projects are often poorly coordinated, inadequately supervised, and ineffective in delivering products of demonstrably high quality.

Making the process of developing software explicit can address many of these problems. Indeed it seems that increasing the rigor and specificity of the definitions of software processes should lead to increased effectiveness in developing software products whose quality is both assured and demonstrable. Carefully and explicitly defined software processes can specify exactly how and when the various components of a software product are to be produced, analyzed, and tested, thereby supporting demonstration of product quality.

More careful and explicit software process definition also facilitates software understanding, development team communication, and project control. But in this paper we emphasize that superior grasp of the process of developing software leads to superior quality in the products developed. We indicate how this results from increased ability to plan quality instilling activities in advance, to specify how they are to be done, to secure support (especially by computers) for executing these activities, and to verify that they have actually been done.

3.1 Software Processes as Software.

The key to seeing how software process rigor and formality effects improved software product quality seems to us to be an appreciation of the nature of software processes as software themselves. Earlier [Ost87] we suggested that processes and applications share many key characteristics. Software processes aim to develop a complex information aggregate (the completed software product), much as application software aims to develop complex information structures. Moreover, just as applications supervise performance of their work through the vehicle of an abstract computing device (consisting of hardware and software), software processes supervise the performance of software development on an abstract device composed of hardware, software, people and organizations. Figure 1 suggests these parallels diagrammatically. Other compelling parallels are put forth in [Ost87, Ost86].

Our conclusion is that software processes can and should be defined precisely and rigorously using pro-

Figure 1 Software processes are a type of software. The software process description is type of code whose execution results in the development of an application software product.

gramming languages. Experimentation with a range of process programming languages [Sut95, KBS90, Kat89], supports this premise. Among the demonstrated benefits of this are: provision of a blueprint for integrating tools to support software development, indication of how development tasks can be parallelized, acquisition of deeper insights into software development as an activity, and coordination of human software workers with support technologies.

Here we are particularly interested in programming processes that lead to the development of software of demonstrably high quality. Earlier we suggested that such quality is best achieved by coordinating testing and analysis with development throughout the stages of the development lifecycle. We indicated that there is agreement about the broad outlines of this approach, but not about the details. Process

programming provides a vehicle for being precise about these details. What is less clear, however, is the rationales for driving the development of software product quality-assuring process code.

3.2 The Spectrum of Software Process Artifact Types.

Thinking of software processes as software helps determine a rationale for integrating quality-assuring tools and technologies. If a software process is software, then it must consist of more than just code. As noted above, quality software consists of a variety of types of artifacts (such as requirements, design, and testcases) as well of demonstrations of their consistency. Thus, quality software processes must also consist of non-code artifacts and demonstrations of consistency as well. Indeed, much software process technology focuses on using modeling and design [MA94, BFG93, ea92]. to create blueprints for process code implementation. But it is requirements specifications that are perhaps the key type of software artifact. Requirements specify what characteristics completed software must have, and are used to make decisions about how product software is to be architected and designed. They are also used to specify the criteria against which completed software code is to be tested. These inculde functionality, speed, accuracy, and safety.

Software processes also have requirements to which they must adhere. For example they have timing requirements that specify the amount of time allowed for completion of the project. They often have robustness requirements that specify process resistance to such unanticipated stresses as loss of key personnel. They also have functional requirements that enumerate the sorts of artifacts to be produced, what their characteristics should be, and the ways in which they must be shown to be consistent with each other.

Software processes also undergo testing, evaluation and evolution. Whenever a software process is used, the outcome is evaluated. Changes are made the process fails to achieve its requirements. Often these failures are recognized while the process is still executing, but often the recognition arises from a static analysis of the process. In any case, process evolution and improvement are to be expected. The view of software processes as complete software products is represented by the diagram in Figure 2.

3.3 The Capability Maturity Model.

The most active and visible work in software process is that centering on the Capability Maturity Model (CMM) at the US Department of Defense Software Engineering Institute (SEI) at Carnegie Mellon University under the direction of Watts Humphrey [Hum88, Hum89]. The CMM is a set of five normative models of software development, augmented by elaborate evaluation instruments and institutions that help determine which of the five models most closely describes the process actually used by a software development organization. The first normative model describes a development process that is essentially chaotic, producing products of poorly determined quality according to an unreliable schedule, without the benefit of careful planning, budgeting, estimation, or commitment. Organizations that fit this model best are classified as Level 1 Organizations, and are said to operate at Level 1.

The second normative model describes organizations that can reproduce their activities and results on successive projects. These organizations generally have thought about their development process, but have not defined it carefully. Level 3 Organizations have defined their process and have used it consistently to produce software in predictable ways and according to predictable schedules. At level 4, organizations are measuring and analyzing their processes, and at level 5, organizations are using processes that automatically measure themselves and use the measurements to automatically improve themselves. SEI evaluations have shown that most software development organizations operate at level 1, although the numbers and percentages of organizations operating at higher levels have been inching up.

CMM evaluations are generally carried out by teams of human assessors who make essentially qualitative judgements about how assessed organizations operate. These judgements are guided by SEI as-

Figure 2 Software process software is more than just software process code. As software itself, software processes are also composed of process requirements, process design, as well as process analyses and execution histories.

sessment guidelines and frameworks that strongly emphasize analysis of documented evidence of how organizations have performed in the past. Thus CMM level assessment is a strong function of the documentation and artifacts produced by previous executions of an organization's process. The use of the CMM to improve organizational process is shown diagrammatically in Figure 3.

A clear overriding goal of the CMM is to instill in organizations a consciousness of the importance of process and a desire to advance to higher levels of process maturity. Thus, the CMM-centered movement is often referred to as the Process Improvement movement. The US military increasingly specifies that it will procure custom-built software only from organizations that demonstrably operate at higher levels of

Figure 3 The CMM is used to study a software development organization's process. CMM evaluations are based on inferences about the nature of the process, drawn from observations of past process executions.

maturity (generally at level 3 or above). This has strongly stimulated software development organizations to improve their processes.

SEI success in popularizing the CMM has interested other organizations in software process assessment. Thus, for example, the International Standards Organization (ISO) is working on ISO 9000, an international standard for software development. The European Software Institute (ESI) is interested in this activity as well, as are software engineering standards bodies around the world. The SPICE project is attempting to harmonize this growing set of process assessment activities and organizations.

Hope that SPICE and other efforts can harmonize these activities seems be based largely on the fact that all center on the notion of Key Process Areas (KPA's). A Key Process Area is an area of

software development endeavor that is particularly critical. KPA's contain collections of practices, called key practices, that contribute to success in the KPA. Some KPA's focus on such practices as planning, estimation, and training. The current CMM version identifies some 28 KPA's, and indicates which KPA is crucial for advancement to which next levels. Different process assessment schemes handle KPA's differently. But there seems to be a clear consensus that the essence of assessment of software processes is scrutiny of how organizations attack various KPA's.

One of the more controversial aspects of the CMM is that it points strongly to certain KPA's as being key in moving organizations to the next highest level. In essence, the CMM uses KPA's to rivet the attention of improvement-conscious organizations on a narrow set of process areas, often to the near exclusion of other areas. Thus, the KPA's most crucial to improvement from lower CMM levels (1 and 2) tend to receive the most attention. It seems widely assumed that improvement from these lower levels results in higher software product quality. While this may be the case, it is not because there are large numbers of KPA's that focus on this as an issue. Indeed, any improvements in software quality result indirectly from other process issues.

3.4 The CMM as a Software Process Test Plan.

Earlier we noted that software processes, like other kinds of software, evolve when they fail to adequately address their requirements. We saw that the CMM supports this sort of process improvement by serving as a guide in assessing software processes. We further noted that CMM-based assessments examine artifacts for evidence of sufficient competence in KPA's. This evidence is generally in the form of records and artifacts produced by past executions of the organization's process.

Thus, we see that the CMM functions essentially as a test plan for software processes. It mandates certain kinds of behaviors as being required if the process is to be assessed at a higher level, and accepts as evidence artifacts produced as outputs from process executions.

While the CMM has had an undeniably positive impact in focusing attention on the need to improve software processes, we can now see some flaws in this approach. We noted above, for example, that quality is best built into software, rather than being tested in at the end. Yet, paradoxically, we see that CMM-based process improvement inherently attempts to improve process quality through post hoc testing of the process. One problem with post hoc testing is that, once the test plan is known, software development tends to be skewed to address key testing criteria. In this case, since the CMM is known in advance, software process improvement efforts tend to be skewed to address those KPA's that are keys in supporting organizational advancement to the next higher level.

Of greatest concern to us, however, is the direction in which the CMM tends to skew improvement efforts. Careful study of the CMM and its KPA's suggests that the key goal of the CMM, and the key dimension along which process improvement is encouraged, is process predictability, and not product quality. With KPA's focusing on prediction, estimation, and planning, the current version of the CMM incentivizes and rewards organizations for becoming increasingly accurate in predicting cost and schedule. As the CMM is produced by the DoD-sponsored SEI this is neither surprising nor inappropriate. Further, suggestions that product quality improvement is strongly correlated with organizational predictability seem both plausible and statistically supported. Nevertheless, it seems important that organizations whose primary concern is for software product quality realize that their primary objective is not the same as the primary objective of the current CMM-based process improvement movement. In the next section we outline a proposal that should interest organizations whose primary goal is assuring software product quality.

4 THE IMPROVEMENT OF SOFTWARE QUALITY INSTILLING PROCESSES.

We believe that the key to systematic improvement in the ability of processes to build in and demonstrate quality in software products is to specify precise software quality demonstration requirements for software processes and to use a CMM-like approach to incentivize improvement along this software process dimension.

4.1 Software Product Quality Requirements.

Earlier we indicated that quality in software is a many-faceted, many-dimensional characteristic. We described failures in functionality, efficiency, safety, and robustness as examples of ways that software could demonstrate lack of quality. Later we suggested that software should be viewed as the product of a software development process. ¿From this viewpoint we see functionality, efficiency, safety, robustness, etc. as software product requirement types.

Thus we can view the demonstration of quality in software as being the process of demonstrating the adherence of product software to specified quality requirements. This explains why different software products require different sorts of quality demonstrations. It also argues strongly for the importance of identifying what those quality attributes must be in advance of the development of the product software. Being precise about specific quality requirements is most important, but it is also quite difficult. It seems, at first glance, quite easy to specify that the software product must never fail to compute required functions, that it must never fail to execute sufficiently rapidly, and that it must never violate safety conditions. On the other hand, simply specifying these conditions leaves unanswered the more important and challenging question of what evidence that the software meets these requirements is to be taken as acceptable. We have already noted that virtually all programs can be subjected to virtually limitless amounts of dynamic testing. At what point should testing activities be considered adequate? How thoroughly should each of the many functions of a program be exercised? How precisely should timing be measured? What procedures should be followed in order to assess a software system's user-friendliness?

These and similar questions seem to us to be process-related. They suggest that it is more fruitful to consider quality determination to be a process, and to consider what the requirements for this process are. Consideration of the requirements for the software product quality determination process seems to lead to more useful conclusions about how to perform software testing and analysis.

4.2 Software Product Quality Determination Processes.

We now consider software product quality determination processes to be types of software. We begin by considering their requirements. This transforms the more traditional question, "What qualities must product software have?" to the more pragmatically useful question, "What demonstrations of software quality will be acceptable as sufficient evidence that product software has the desired qualities?" OR better yet, "What do I wish to learn about my program, and how will I know when I have learned it?" Remarkably, many software testing and evaluation processes begin, and end, without any clear enunciation of even such primitive goals. Small wonder that so many testing activities end unsatisfactorally. Without specified requirements for this process it is impossible to determine whether it has succeeded or failed.

Once quality determination process requirements have been specified it is possible to design the process of making the determinations. Presumably the determinations are to be made by a skillful integration of such techniques as dynamic testing and static analysis applied to the software artifacts produced as by the development process. Different requirements will lead to different configurations and integrations. This is reassuring as it explains the observed phenomenon of widely differing approaches to software

quality determination employed in actual practice. This diversity should be expected as a consequence of different requirements.

Quality determination should then execute in parallel with product development. Evidence suggests that concentrating quality determination towards the beginning of the development process is most effective, but experience shows that most development activities will concentrate quality determination towards the end. In any case, it is most important that the quality determination process itself be evaluated as it executes. While it is hoped that specifying requirements for the process, and designing it carefully, will lead to more successful quality determination processes, it must be expected that they may still fail to meet their requirements, and will require modification, evolution, and improvement. The need for such modification, and the exact nature of the improvements needed, cannot be determined without careful evaluation of the process itself.

4.3 An Example.

The preceding discussion is clarified by an example. Suppose an aircraft navigation system must be built. Central to the navigation system is software responsible for reading location, altitude, and speed data from sensors, comparing that data to routing vectors, and sending realtime instructions to flight control systems to determine the course of the plane. Clearly this software has functional requirements. In addition, however, it has stringent performance requirements needed to assure that modifications in direction are timely. There are safety requirements to assure that the software never misdirects the plane disastrously. There are also user interface requirements, accuracy requirements, and others.

These requirements must be used to establish baselines against which to evaluate the quality of the software to be built. It is clearly necessary, but definitely not sufficient, to simply require that the eventual software system will adhere to these requirements. Some of these requirements will be of greater importance than others. Thus, assurances that they have been met will be of greater consequence, and confidence of the effectiveness and accuracy of the processes for assuring them will need to be higher. It seems more direct and straightforward to underscore and operationalize these differences in importance and priority by elaborating these basic product quality requirements into quality determination process requirements.

Thus, for example, functions that compute the location of the plane accurately may be considered more important than functions that update cockpit displays accurately. In that case greater assurance of the effectiveness of the quality determination process would be warranted for the former functions than for the latter. This might be made quite clear and concrete by indicating different degrees of assurance of the absence of faults in the computation of these different functions. In the case of less important functions a testing regimen requiring a certain more modest level of coverage might be mandated. Once the testing level has been established, testing testing coverage should be measured on an ongoing basis, and testing should continue until it has been reached, at which point it should terminate. For more critical functions, however, it would seem reasonable to require more assurance, and to respond by designing a quality determination process that also included more rigorous forms of analysis of software functionality. These attempts at more rigor should likewise be monitored, and failure to complete them would require more urgent response.

Similarly, considering the requirements for the process of determining acceptable performance leads to important sharpening of the basic product performance requirements. It may be that new position and direction information must be determined every millisecond without fail. Or it may be that it is permissible to be late with this information occasionally. In the latter case, it is important to define just what is meant by "occasionally," and it might suffice to respond by designing a dynamic testing process that measures the percentage of cases in which results are computed late. The former case implies that the quality determination process must deliver ironclad assurances of timeliness. In this former case, testing may be a reasonable way to start off, but some sort of static analysis activity would be necessary later on. Further, the requirement for ironclad assurance dictates that the process of enunciating and completing

analytic proofs would have to go to completion. Failure of the quality determination process to do so would have to be detected as part of the ongoing evaluation of the process, and would require decisive response: either modification of the process, or modification of the requirements for ironclad assurance.

Specific safety properties must be stated precisely. Once enunciated, the designer of the quality determination process would have to devise effective demonstrations that they have been satisfied. As in the preceding discussion, this process will require ongoing evaluation to assure that it does not fail to deliver the ironclad assurances required. It is doubtful that dynamic testing can ever deliver such assurances, but static dataflow analysis often can. For example, safe operation of the plane might require that newly computed position and direction data never be be overwritten before being used to compute new heading data. This can be readily transformed into an event sequence specification that can then be the basis for static dataflow analysis. If attempts to complete this analysis fail, then either analysis efforts must be strengthened, or the safety requirement reconsidered.

In all of the above cases, we see that the need to operationalize the process of determining product quality, and to continually compare success in doing so to specified process requirements, has led to the ability to be more specific about how to design the process. This in turn leads to the ability to be more specific in the details of which testing and analysis approaches to use, and how to integrate them. An important consequence of this approach is that it makes very clear the fact that one should neither expect nor seek the emergence of one standard testing and analysis process for all software. Different software systems have different product requirements. These in turn give rise to different quality determination processes, and they give rise to differences in the specifics of how to integrate testing and analysis into the development lifecycle.

On the other hand, all such quality determination processes share the need for continuous improvement. We believe that a CMM-like approach can be quite useful in guiding improvement efforts.

4.4 Software Product Quality KPA's.

The foregoing discussion has shown that one should expect quality determination processes to evolve over time. Quality determination processes have uncertain outcomes because of uncertainty about the inherent quality of the software being assessed, unfamiliarity with how to state requirements, perform design, and encode such processes effectively, and because of the immaturity of current testing and analysis tools. Thus quality determination processes must be the subjects of continuous improvement. This suggests considering using a CMM-like approach to do this.

Indeed we believe that quality determination processes should progress from being chaotic, to repeatable, to defined, to measured, and ultimately to self improving. Thus, the basic structure of five normative models seems applicable here. Further, the use of Key Process Areas (KPA's) would seem to be appropriate here as well. It seems important to reflect on what these KPA's might be. Certainly planning for software product quality determination as a separate and separable activity would have to be one such key process. Measuring testedness, observing analysis failures, reviewing product quality determination progress, and developing process architectures and designs all seem to be likely candidates for KPA's as well. This seems to be an important area for further investigation.

Finally, we can now return to our earlier observation that the directions in which the current CMM is driving improvement are not directly towards product quality. As KPA's seem to be the vehicles for setting such directions, it now seems that identifying KPA's specific to product quality determination might be just the needed vehicle. Current discussions of the inclusion of software product quality KPA's in the next version of the CMM is encouraging. But organizations focused on software product quality improvement need not wait for CMM revisions. They should be able to begin identifying and institutionalizing key practices immediately. Superior quality in their products should ensue.

REFERENCES

[AGB+77] A.L. Ambler, D.I. Good, J.C. Browne, W.F. Burger, R.M. Cohen, C.G. Hoch, and R.E. Wells. Gypsy: A Language for Specification and Implementation of Verifiable Programs. In *Proceedings of ACM Conference on Language Design for Reliable Software*, pages 1–10, March 1977. Published as *SIGPLAN Notices, Vol.12,No.3.*

[BFG93] Sergio Bandinelli, Alfonso Fuggetta, and Sandro Grigolli. Process Modeling in-the-large with SLANG. In *Proceedings of the Second International Conference on the Software Process, Berlin, Germany*, pages 75–83, February 1993.

[Bjo87] D. Bjorner. On the Use of Formal Methods in Software Development. In *Proceedings of the Ninth International Conference on Software Engineering*, pages 17–29, March 1987. Published by IEEE Computer Society Press.

[Cla76] Lori A. Clarke. A System to Generate Test Data and Symbolically Execute Programs. *IEEE Transactions on Software Engineering*, SE-2(3):215–222, September 1976.

[DC94] Matthew Dwyer and Lori Clarke. Data Flow Analysis for Verifying Properties of Concurrent Programs,. In *ACM SIGSOFT'94 Software Engineering Notes, Proceedings of the Second ACM Sigsoft Symposium on Foundations of Software Engineering 19(5)*, pages 62–75, December 1994.

[ea92] Richard J. Mayer et al. IDEF Family of Methods for Concurrent Engineering and Business Reengineering Applications. Technical report, Knowledge Based Systems, Inc., 1992.

[How78] W.E. Howden. Introduction to the Theory of Testing. In E. Miller and W.E. Howden, editors, *Tutorial: Software Testing and Validation Techniques*, pages 16–19. IEEE, New York, 1978.

[How87] W.E. Howden. *Functional Program Testing and Analysis.* McGraw-Hill Series in Software Engineering and Technology, 1987.

[Hum88] Watts S. Humphrey. Characterizing the Software Process: A Maturity Framework. *IEEE Software*, pages 73–79, March 1988.

[Hum89] Watts S. Humphrey. *Managing the Software Process.* SEI Series in Software Engineering. Addison-Wesley, August 1989.

[Kat89] Takuya Katayama. A Hierarchical and Functional Software Process Description and its Enaction. In *Proceedings of the Eleventh International Conference of Software Engineering*, pages 343–353, 1989.

[KBS90] Gail E. Kaiser, Naser S. Barghouti, and Michael H. Sokolsky. Experience with Process Modelling in the MARVEL Software Development Environment Kernel. In Bruce Shriver, editor, *Proceedings of the Twenty-Third Annual Hawaii International Conference on System Sciences, Kona, Hawaii*, volume II, pages 131–140, January 1990.

[Kin75] James C. King. A New Approach to Program Testing. In *Proceedings of the International Conference on Reliable Software*, pages 228–233, 1975.

[MA94] Carlo Montangero and Vincenzo Ambriola. OIKOS: Constructing Process-Centered SDEs. In Anthony Finkelstein, Jeff Kramer, and Bashar Nuseibeh, editors, *Software Process Modelling and Technology*, chapter 6, pages 33–70. John Wiley & Sons Inc., Taunton, Somerset, England, 1994.

[OF76] Leon J. Osterweil and L.D. Fosdick. DAVE — A Validation, Error Detection, and Documentation System for Fortran Programs. *Software Practice and Experience*, 6(4):473–486, October 1976.

[Ost86] Leon J. Osterweil. A Process-Object Centered View of Software Environment Architecture. In R. Conradi, Didriksen T, and D. Wanvik, editors, *Advanced Programming Environments*, pages 156–174, Trondheim, 1986. Springer-Verlag.

[Ost87] Leon J. Osterweil. Software Processes are Software Too. In *Proceedings of the Ninth International Conference of Software Engineering*, pages 2–13, Monterey CA, March 1987.

[RAO92] Debra J. Richardson, Stephanie Leif Aha, and T. Owen O'Malley. Specification-based Test Oracles for Reactive Systems. In *Proceedings of the Fourteenth International Conference on Software Engineering, Melbourne, Australia*, pages 105–118, May 1992.

[RT86] Debra J. Richardson and M.C. Thompson. A Formal Framework for Test Data Selection Criteria. Technical Report 86–56, Computer and Information Science Department, University of Massachusetts

at Amherst, Amherst MA 01003, November 1986.

[Ryd74] B.G. Ryder. The PFORT Verifier. *Software — Practice and Experience*, 4:359–378, 1974.

[Sut95] Sutton, Jr., Stanley M. and Heimbigner, Dennis and Osterweil, Leon J. APPL/A: A Language for Software-Process Programming. *ACM Transactions on Software Engineering and Methodology*, 4(3):221–286, July 1995.