

## **SWEBOK Review Comments**

### **Trial Version 1.00 (downloaded 6/4/03)**

Douglas Hoffman BACS, MSEE, MBA, ASQ Fellow, ASQ-CSQE<sup>1</sup>  
[Doug.hoffman@ieee.org](mailto:Doug.hoffman@ieee.org)      [www.SoftwareQualityMethods.com](http://www.SoftwareQualityMethods.com)

#### **Summary:**

Outlining and describing the current state of the practice is an excellent undertaking, but at the same time it can be misleading and dangerous. My broad experience with computer system manufacturers, software companies, established organizations, startups, embedded systems, operating systems, middleware, applications, tools, regulated and unregulated industries, and more has given me a unique perspective on what constitutes good, useful, helpful practices for a huge range of circumstances. I've found that nearly all documented techniques are extremely valuable for some context, but those same techniques are counterproductive in other contexts. I believe that SWEBOK must identify the good practices while allowing for contexts for which they are inappropriate. Indicating that they are "best practices" (and thus always applicable) is harmful to the profession.

My biggest concern with a body of knowledge (BOK) like SWEBOK is that it tends to exclude many valuable emerging and niche elements and at the same time indicating that some specific techniques should be universally practiced. Worse, published BOKs are used for certification of professionals and become the basis of professional malpractice when techniques described in them are not blindly followed. A BOK is only valid for certification when the contents are universally applicable or when the applicable contexts are clearly understood and articulated. A technique that is applicable and useful for software used in controlling an airplane may not be applicable or useful for software used in a DVD player.

Unfortunately, I do not see enough context specific qualifiers with practices in the SWEBOK. (Indeed, I'm not sure we computer scientists understand enough about computers and software to describe relevant context characteristics.) Given a choice, I would vote "No" on accepting the SWEBOK, not because it is necessarily bad or wrong, but because it will mislead and possibly confound progress in computer science. It's the first step toward legislating the one way all software will be "engineered."

I began reviewing the SWEBOK intending to take a cursory look at the sections I am most proficient with and providing suggestions and possibly a few critiques. I spent several hours going into the first two chapters before skipping to the chapters on Testing and then Software Quality. I was encouraged by the explicit recognition that different organizations, users, and products require different techniques in both chapters. But, I was discouraged by the many deficiencies in the Testing chapter and the gaping holes in the Software Quality chapter.

I am shocked by the fact that no reference is made to ASQ's CSQE, if even to criticize it. Are the drafters of the SQEBOK really ignorant of the existence of a sister society's related Software Quality Engineering Body of Knowledge, or have they chosen to ignore it because it might be inconvenient or at odds with SWEBOK?

---

<sup>1</sup> Douglas has over 30 years experience as a practitioner and instructor in computer science since earning his BA in Computer Science. He has extensive experience in quality assurance, testing, test automation, requirements, development processes, and project management in particular.

***Specific comments and examples:***

**Chapter 1; Introduction to the Guide**

We may desire to have Software Engineering, but the state of the practice does not support it. We have not agreed upon the meaning of basic terminology or discovered any underlying principles from which to "engineer" software. I can understand and agree with the desire to consider software development to be an engineering discipline, but that doesn't change the fact that the current state of the science doesn't support it.

How large are programs? What constitutes a defect in a program? How complex is a program. What does it mean if one program is more complex than another? (Does the more complex software have more defects? Is it bigger? Is complexity due to the problem being solved or programming language or program style?) How do we stress software? We haven't agreed upon the answers to these questions because we don't yet have a grasp of the fundamentals. This isn't about wanting standards or consistency, it's about the state of the science in today's computer science.

I contend that the current state of computer science is roughly comparable with biological science 150 years ago. Many of the greatest scientific minds of the day were drawing pictures, observing, and doing scientific experiments in order to figure out the fundamentals. They were very far from biological engineering. No amount of wanting, debating, or legislating could change the state of the science. Only continuing scientific observations and experimenting could advance the state of the science. Pretending to have biological engineering would have slowed or stopped the progress of scientific learning. Of course every doctor knew disease wasn't caused by invisible animals and they knew a person couldn't become infected by mear touch. They had all practiced medicine all their lives without washing their hands. Anyone who said different was obviously a crazy heratic.

It is premature to speak of certification to professional standards in computer science beyond ethics and scientific methods. Current "best practices" in computer science are guesses based on imperical evidence. Some are "best" in some circumstances. I know of none that are always best.

**Chapter 2; Software Requirements:**

**Section 2.4; Requirements engineering in practice**

An expressed assumption is that the waterfall process model is applicable to requirements. This is not always true. Many, if not most of my clients and projects over the years have been based on creating a new product or type of product. In all of these cases we have not been able to successfully determine the requirements beforehand. In these software development processes requirements elicitation and analysis are iterative and ongoing, the requirements specification is minimal and relegated to change management, and where requirements validation is the same as

elicitation. We discovered the requirements based on user feedback, using an ongoing process of rewriting the requirements and redesigning the systems. We used the product prototype as the requirements and specifications. Waterfall methodologies always failed to meet the user's needs and the products were commercial failures. Many of the products whose requirements were discovered were very successful. The Palm Pilot is one example of success, while Apple's Newton failed. Palm used iterative development, Apple used waterfall.

Requirements documents are not always able to be validated. When the documents are used as starting points or when the product is truly new, validating the documents themselves isn't useful. The documents may be found to reflect current stakeholders' requirements, but we may know further changes are required (we just don't know what those changes are yet). We learn nothing of value from validation. We want instead to find out what new or changed requirements could be useful. Rather than validation of the requirements we want to question their usefulness.

## **Section 2.5; Products and deliverables**

Requirements traceability is a reasonable technique for many projects, but it assumes relative stability of the requirements, which is not always true. When requirements are changing the cost of maintaining the traces can overwhelm the project and unreasonably restrict the product. This is an artificial constraint due to forcing an unnecessary technique and artifact on the project.

Formal requirements documents and requirements tracking are not required for successful software development. Assuming as fact that successful, mature organizations require traceability does not make the practice fundamental to all organizations. Dynamic start-up companies are successful without traceability and sometimes even without written requirements. These software companies succeed and are proof that requirements management and particularly the traceability are not required. Their later use of documented requirements and requirements management techniques merely give us a clue that maybe the techniques are sometimes useful.

## **Chapter 5; Software Testing**

### **Section 1; Introduction**

Testing is also done for identifying useful paths for accomplishing desired tasks. Alternative paths may be chosen so users can be successful in spite of program errors. Instead of testing to find any and all errors, this testing is for the purpose of finding any path to success. This is critically important for acquired software, software tools, middleware, and operating systems.

### **Section 2; Definition of the Software Testing Knowledge Area**

By carefully choosing a definition for software testing you have eliminated and limited the scope of many important and valuable types of tests. Specifications are always incomplete, so we don't necessarily have completely specified expected behavior. The software behavior is due not only to what a software developer wrote, but factors such as the operating system, hardware platform, other software running at the same time, compilers, code and subroutine libraries, and system

environment. Much of my published work has been done on test automation and test oracles to create classes of tests that go far beyond specified expected behaviors.

## Expected

Expected results in observed outcomes do not need to be known in order to have useful tests. Heuristic test oracles<sup>2</sup> describe a class of tests for which the outcome may be unclear. Pass/Fail/Investigate are possible useful test outcomes. [This is also relevant to **Section A2; Theoretical foundations**]

Expected results are not known in many situations where exploratory testing takes place. (Exploratory testing being defined as simultaneous learning, planning, and testing software without reliance upon specifications.) In commercial software organizations the specification is not updated before coding of changes, and therefore the expected results are not known. Many successful software companies do not create or maintain comprehensive specifications.

I have created many automated software tests that successfully find defects but which do not specifically search for specific errors or particular results. Noel Nyman at Microsoft has written about “dumb monkey” automated tests which blindly generate keystrokes<sup>3</sup>. These tests find defects that crash applications without depending upon any knowledge of expected behavior.

## Section 2.1; Conceptual Structure of the Breakdown

Although one of the testing aims is to expose failures, the majority of test techniques are not based on equivalence classes. Random, Exploratory, Use-Case, and many other techniques do not use equivalence class concepts.

Characterizing *the leading principle underlying test techniques* as being based on a) *as systematic as possible*, b) *a representative set of program behaviors*, and c) *subclasses of the input domain* would appear to be based on either ignorance or rigidity. Many good tests are not systematic. Sometimes systematic testing misses even gross errors because of their very nature. More defects will be found by causing variations (especially random variations) than will be found by doing the same tests the same way.

Although the majority of textbooks on software testing, and possibly the majority of tests, may be based on equivalence classes, that does not make it the foundation of testing. Very valuable tests and sophisticated defects are possible once a tester gets beyond the limited concepts of domains and classes.

Many of the tests I have implemented are not focused on representative sets of program behaviors, but rather on choosing interesting combinations and series of behaviors. Many excellent tests find defects by exploring beyond boundaries or choosing combinations that are

---

<sup>2</sup> Hoffman, Douglas, “*Heuristic Test Oracles*,” *Software Testing and Quality Engineering Magazine*,” Volume 1, Issue 2; March/April 1999.

<sup>3</sup> Noel Nyman, *GUI Application Testing with Dumb Monkeys*.

purposely unexpected or illogical. Stress based testing is based specifically on nonrepresentative behaviors.

It is particularly surprising that the SWEBOK would focus on subclasses of input domains and overlook other fundamental subclasses such as output domains, programmatic interfaces, devices, environments, and internal variables. We should see emphasis on advanced techniques rather than recommendations that focus practitioners on simple approaches.

I agree wholeheartedly that we *must include other [objectives like] reliability measurement, usability evaluation, [and] contractor's acceptance*. However, you then admonish not to confuse test objectives and techniques and specifically exclude possible objectives such as branch coverage. I have worked in regulated, contract based, and litigation sensitive organizations where a goal was precisely that – 100% code or branch coverage. This is likely to be even more important to organizations when the SWEBOK is accepted and code coverage becomes a best practice (legally).

The pushing of test automation has caused tremendous harm and dramatic decrease in the test effectiveness. We should not blindly support or encourage test automation for cost effectiveness. I have discussed this in my paper on automation cost benefits<sup>4</sup>.

## **Section A2; Theoretical foundations**

### **• The oracle problem**

When we decide whether a program behaved correctly (by whatever oracle) we are always sampling only a small portion of the actual outcomes. It should be noted in the SWEBOK that even though an algorithm produced an apparently correct result, there are an uncounted number of possible things that could have occurred that we did not detect. Memory leaks, database corruption, file system errors, security violations, network bandwidth, program timing, and device interference are all examples of common classes of errors we seldom check for. [Dijkstra was right in more ways than he thought of at the time.]

[See also the notes above under **Section 2; Expected** for a relevant discussion of pass/fail/investigate]

## **Section B; Test Levels**

The list of objectives (types of tests) does not include several types of tests I have employed:

- Load (providing a nominal load for background, resource consumption, or collision/interference testing)
- Life (running a program or system for weeks to test for resource exhaustion, overflow conditions, or other failures that require many, many iterations)
- Worst Case (combinations of inputs and environmental factors)
- Perfective (tests for added capability)
- Exploratory (discovery/investigation into new areas or using new approaches)

---

<sup>4</sup> Hoffman, Douglas, “*Cost Benefits for Test Automation*,” STAR West, 1999.

- Random-Input (automated)
- Certification (meets a given standard, passes a set of tests designed to show compliance with a specific standard)
- Real Time (activities requiring timing)
- Collision (competing with other software for resources – record locking, memory usage, shared devices, etc.)

## Section B2; Objectives of Testing

- **Regression testing**

There are two additional definitions I have found to be common; testing to verify that a fix was effective (a.k.a., “bug regression”), and testing to check that a particular error has not recurred (usually as a suite of tests for all previous fixes).

## Section C; Test Techniques

The term Black-Box testing predates integrated circuits, although it is from the field of electrical engineering (the authors are apparently showing their age). I do agree that the definitions presented are reasonable.

### Section C1.1; Based on tester’s intuition and experience

I believe it would be a disservice to ignore or support unplanned, unthoughtful banging on a keyboard (*ad hoc* testing). However, the authors are apparently ignorant of the difference between *ad hoc* and *exploratory* testing. I cannot understand how anyone who claims to be professional in a field can be unaware of the existence of techniques that have been presented at professional conferences for the past five years. I also cannot understand how professionals can pass judgement on approaches they clearly don’t understand. Professionals learn techniques of their trade, including techniques they may choose to avoid, so they can make rational decisions based on knowledge. Professionals do not profess knowledge in areas they don’t understand.

I agree with the characterization of *ad hoc* testing as generally less effective than systematic testing. At the end of a day of *ad hoc* testing, the tester cannot adequately describe what they’ve covered. We cannot distinguish between excellent product quality and poor testing.

For *exploratory* testing we can not only describe what has been tested, but there is a partial map of product capabilities discovered and pointers to possible areas for follow up testing. This is possibly the most effective mechanism for finding defects in a software product. Over the past ten years James Bach ([www.satisfice.com](http://www.satisfice.com)) has generated a great deal of material explaining the process and teaching how *exploratory* testing can be done.

### Section C1.2; Specification-based

Random testing does not need to be purely random or based upon the operational profile. I have generated many automated tests using biased random values and oracles. In several cases the

biasing is based on input to the test, so the tests could be easily pointed toward or away from specific features or values. The power in these tests comes from exercising huge numbers of values or combinations of values, finding unknown (often unknowable) classes and/or boundaries that other specification based approaches will reliably, systematically miss.

### **Section C1.3; Code-based**

It is arguable whether Mutation testing provides any value as a tool for evaluating test coverage, much less for generating tests. There is no basis for believing the underlying assumptions, there is no reason the generated mutations have any relationship to actual errors, and generating a high number of mutants is impractical in any but trivial programs. I do not see any difference between this and debugging.

### **Section C2.2; White-box techniques**

I have done work with several model based random tests, which rely upon knowledge of the code and therefore are white-box tests. These tests have ranged from dumb (very little verification of responses) to intelligent (able to detect correct responses to valid input and expected error responses for other input). Noel Nyman has also published papers on this approach<sup>5</sup>.

### **Section D; Test related measures**

I have used and taught software metrics all through my career. I do not believe we have scientific basis for any of the software metrics I have encountered throughout my career<sup>6</sup>. When used for purely observational purposes, IEEE 982.1 measures have sometimes helped provide insight into products and processes. However, when used for control purposes, I have yet to see a software metric that wasn't subject to tampering. At best they have distorted processes and misled management. I believe a the SWEBOK must include a strong caveat explaining that metrics do not necessarily tell us what we really want to know.

The problem lies in our application of convenient numbers without a scientific foundation. Cem Kaner has published several articles addressing some shortcomings and approaches for software metrics ([www.kaner.com](http://www.kaner.com)). Take for example one of the best known and least controversial metrics, fault density. It is in common use, and takes the form of the ratio of number of defects to the size of code. For a given product and defect database we can easily compute the ratio (in several ways), but what does it mean?

If the density goes up, is there necessarily a degradation of quality? This could happen for many reasons unrelated to the code itself; new tests have begun and there are new defects being reported, or dead code has been removed, or a module has been streamlined (fewer lines of code). These are examples of higher defect density without changing code functionality or defectiveness.

---

<sup>5</sup> Noel Nyman, "GUI Application Testing with Dumb Monkeys"

<sup>6</sup> Hoffman, Douglas, "The Darker Side Of Software Metrics," *PNSQC 2000*

If the density goes down, is there an increase in quality? This could also happen for many reasons unrelated to the code itself; testing is curtailed (so the defect count is reduced by attrition), or the size of the code has increased (sometimes without any change in functionality), or defects are reclassified so they are no longer counted. These are examples of lower defect density but without improvement in code defectiveness.

What if the people doing the testing are ineffective, or only report reliably repeatable defects. This makes the metric look better, but does not make the software less defective.

These examples are real, from companies and projects I have worked on. The underlying problem is that there isn't a direct, scientific model for program quality. We can count reported defects or compute function points, but we have no scientific justification for asserting that either of them represents a program characteristic. We certainly have no scientific model to relate the measures to program quality.

There is value in investigating, observing, postulating, comparing, contrasting, and debating models for software, defects, code size, defects, etc. Today, however, we must take them on faith and empirical evidence alone. Failure-count and time-to-failures models have been postulated, and the faithful can produce evidence of their validity. Unfortunately, many people don't question the assumptions and assume that any model that can be expressed is valid.

## **Section E; Managing the Test Process**

### **Section E1; Management concerns**

- **Test documentation and workproducts**

The vast majority of successful organizations I have worked with use far less than the IEEE 829 specified test documentation. Very few organizations have complete requirements before testing begins. The only cases of successful projects in my experience with the prescribed test documentation and work products were contract software or regulated industries. It would be professional malpractice to force all software development organizations to use the prescribed documentation.

- **Internal vs. independent test team**

I have successfully worked with developers who acted as testers for their own code. Each of them had unique knowledge that meant that no one else could test their code. There could be ethical and possibly practical reasons for avoiding developers testing their own code, but that does not mean team members who are directly involved in code development cannot perform the task.

- **Effort estimation and other process measures**

[See the discussion of metrics in **Section D; Test related measures** above.]

## **E2; Test activities**

- **Test case generation**

It is neither necessary or always best to specify the expected results for each test. We may be operating from incomplete information and therefore part of the purpose of a test is to discover what the software does. Many times I have written negative test cases to discover whether the software does something reasonable or acceptable. In many of those cases the behavior was expected to change over time, so specification of expected results beyond “does something acceptable” is not useful.

- **Test environment development**

In many situations the development environment is not the target environment, so the test team is required to be compatible with the users’ environments – not the development environment. The recommendation should be for the test environment(s) to be identified and controlled.

- **Execution**

In commercial environments the tests are almost always performed by the quality assurance personnel who designed and implemented them.

Limiting testing to that which is so well documented that someone else could replicate results is clearly limited thinking. Product quality often suffers when testing is limited to predefined tests. Time spent generating and maintaining detailed documentation of the test plans, test specifications, etc., is time not spent doing testing. In commercial software development the purpose of testing is to figure out if the software is ready for release. The emphasis in testing shifts as our knowledge changes. New tests are created as required. Many tests are run once, results recorded, and often then thrown away (usually when no problems were uncovered). There is a clear cost trade-off between formal planning, formal documenting, and testing.

When the test workproducts are products for delivery (to customers, to regulators, to maintenance staff, etc.), all of the test execution components identified in the SWEBOK may be appropriate. However, in the majority of commercial situations the workproducts are for the benefit only of the testers, and minimum efforts should be wasted on formalisms that reduce test time or test effectiveness. Yes, the testing must be organized and managed, the results properly recorded and tracked. But replication is not always desirable, and procedures not always clearly documented.

- **Defect tracking**

I agree that it would be nice if we analyzed when errors were introduced, their cause, when they could have been first observed. My experience has been that commercial organizations do not reliably record that information. These things should not be stated as necessary, as many successful organizations do not do them.

## Section 4; Breakdown Rationale

I am very surprised by the statement “The position taken in writing this document has been to include any relevant topics in the literature, even those that are likely not considered so relevant by practitioners at the current time.” There are many topics that have been left out or renounced. The claim of inclusion is clearly untrue (or the SWEBOK has been authored by people ignorant of the current state of the practice in software testing). Test First development, Exploratory Testing, Minimal Effective Documentation, Automated Random Tests, and the nine levels of tests I listed above have all been excluded.

## Chapter 11; Software Quality

### Section 1: Introduction and Definition of the Software Quality Knowledge Area

Why is the Software Quality KA limited to product related quality and V&V? The American Society for Quality has defined a BOK for Software Quality Engineering (<http://www.asq.org/cert/types/csqe/bok.html>) that lists seven areas of knowledge:

1. General [Quality] Knowledge, Conduct, and Ethics
2. Software Quality Management
3. Software Engineering Processes
4. Program And Project Management
5. Software Metrics, Measurement, And Analytical Methods
6. Software Verification And Validation
7. Software Configuration Management

This chapter only addresses V&V. The remaining topics are overlooked or referenced. Although two of the CSQE topics are covered in some depth elsewhere in SWEBOK (Software Engineering Processes and Software Configuration Management) and four are covered briefly by reference (Software Engineering Management, Engineering Design, Engineering Methods and Tools, and Metrics). SWEBOK overlooks general understanding about quality, management of the quality functions in the development process, management of the software quality organization, and organizational change management. The Software Quality area of SWEBOK seems to be wholly inadequate in covering software quality.

#### 2.5.1. Fundamentals of Measurement

I see an excellent start to the metrics; describing the differences between *nominal*, *ordinal*, *interval*, and *ratio scale*, and the phrasing of the definition of measurement as “*the assignment of numbers to objects in a systematic way to represent properties of the object.*” After that, though, I suspect some pseudo-science is applied.

I’m not quite clear about the of the classification of defects discovered per module as being a ratio value.

1. What does it mean when 10 defects are reported for module 1 and 20 reported against module 2?
  2. Is module 2 twice as defective?
  3. How much effort has been put into testing module 1 or module 2?
  4. Module 3 only has one defect reported (it won't start). That makes it the best of the bunch.
  5. Is module three 10 times better than module 1?
- These counts are not ratio. They are nominal.
- The counts only represent what we've recorded. This may or may not be meaningful with regard to module defectiveness. (Almost always not.)

IF defect counts represent the actual defectiveness of a module:

1. What actual property of the program does it represent?
  2. If I discover and report a new defect in a module that was not changed, how did it become more defective?
  3. How could it have the property of 3 defects yesterday but 4 today?
  4. Is a different module with a defect count of 10 more defective? Does it matter how much testing has been done? Does the size matter
- The defect count is a convenient number readily at hand. It means no more or less than the number of reports in our data base.
- This is a nominal value.

You immediately change the discussion to defects per function point (a computed ratio) and give arithmetic examples. The example assigns 10 to module1, 15 to module2, and 20 to module3. (I assume you're computing a defect density, but not using the term.)

1. What actual property of the program does defects/FP represent?
  2. If we add/remove code to the program, does the value always increase/decrease? Does the property always change the same way?
  3. If I discover and report a new defect in a module that was not changed, how did it become more defective?
  4. What is the program property by which module3 is twice module1? (What does it mean?)
  5. What if we actually found all of module3's defects but only 1/6 of module1's?
  6. How could our metric really represent a property of the object but be wrong?
  7. How do we know all the defects in the modules have been found/reported?
- The ratio of defect count to FP is a convenient number. It means nothing more or less than the number of reports in our data base divided by an artificially generated value for program size.
- The ratio is not a meaningful value that measures any program attribute because of variability, inaccuracy, and biases of the defect count.

## Section 2: Breakdown of Topics for Software Quality

The definitions provided for Software Quality are limited and the second, "*the efficient, effective, and comfortable use by a given set of users for a set of purposes under specified conditions*" seems to be untestable. How are *efficient, effective, and comfortable* to be measured? What of software products that are used by a different set of users or under different conditions? The choice of these biased definitions of Software Quality stems from the desire to force fit quality to

be “conformance to requirements.” Getting the requirements right to start with and keeping them controlled is an excellent, high quality approach, but, in my experience<sup>7</sup>, as often as not we do not know the requirements beforehand because we have been inventing software products that haven’t existed and aren’t simply coding exercises or contract deliverables.

A better definition of quality might be “Quality is value to some person.”<sup>8</sup> The trick then becomes identifying what is valuable and for which users. Validation is where we really check for quality. Because the SWEBOK has adopted the convenient “conformance to requirements” definition for quality, the remainder of this chapter is missing substantial software quality elements.

By narrowly defining software quality as conformance to requirements, the authors have ignored the majority of software development practices who do not use waterfall or strict adherence to predefined requirements. The rest of the chapter suffers greatly from these omissions. Refer to ASQ’s CSQE BOK.

## **Appendix B; A List of Related Disciplines**

The authors of the SWEBOK appear to be unaware of ASQ-CSQE BOK, as it is not mentioned anywhere in the SWEBOK and many of the topics listed in this established BOK have been overlooked. [See my comments on **Chapter 11; Section 1.**] This is a huge hole that indicates either the creators of SWEBOK are ignorant or they have explicitly avoided reference for unknown reasons.

The only mention of disciplines related to quality are:

- Project Management; Project Quality Management and
- Systems Engineering; Process; Systems Quality Analysis and Management

Taken in combination with the shortcomings of Chapter 11, I am shocked to find the SWEBOK is woefully lacking and certainly not worthy of scholarly consideration at this late date.

## **My conclusions:**

Nearly all the software run on nearly all the computers in the world was developed without the formalisms of the SWEBOK. The PC operating systems, applications, middleware, data communications, peripheral controls – the software from companies like Microsoft, Adobe, Oracle, and Hewlett Packard, Sun Microsystems, Apple Computer – all the flavors of UNIX, web browsers, JAVA – they were all invented with development processes and techniques condemned or ignored by SWEBOK. The fact is that good quality software can be created,

---

<sup>7</sup> I’m speaking of many successful organizations such as Hewlett-Packard, Sun Microsystems, Adobe Systems, and Palm Computing.

<sup>8</sup> Weinberg, Gerald, “The Psychology of Computer Programming,” Silver Anniversary Edition, p.2.i (Dorset House, 1998)

software engineering can be performed entirely outside the SWEBOK framework. It is a shame the SWEBOK developers did not know about or embrace so many of the techniques actually employed in effective software engineering practices.

I agree with the ACM appraisal that the SWEBOK started with a fundamentally flawed approach. The SWEBOK is still fundamentally flawed.

I teach courses in software testing, software quality assurance, and software project management. SWEBOK is not a good reference point for them. I will only use it if forced to do so, and then I will teach it as the dogma of a group of well meaning but limited people with an agenda other than moving software engineering forward.