ELSEVIER

The Journal of Systems and Software 62 (2002) 59–70

The Journal of Systems and Software

www.elsevier.com/locate/jss

Fundamental principles of software engineering – a journey

Pierre Bourque [a,*], Robert Dupuis [b,1], Alain Abran [a], James W. Moore [c,2], Leonard Tripp [d,3], Sybille Wolff [e,4]

[a] *Département de Génie Électrique, École de Technologie Supérieure, 1100, rue Notre-Dame Quest, Montréal, Québec, Canada H3C 1K3*
[b] *Université du Québec à Montréal, P.O. Box 8888, Succ. Centre-Ville, Montréal, Québec, Canada H3C 3P8*
[c] *The MITRE Corporation, 7515 Colshire Drive, McLean, VA 22102-7508, USA*
[d] *Boeing Company, Seattle, WA, USA*
[e] *SAP Labs, Montreal, Quebec, Canada*

Received 30 December 1999; received in revised form 25 March 2000; accepted 8 August 2001

Abstract

A set of fundamental principles can act as an enabler in the establishment of a discipline; however, software engineering still lacks a set of universally recognized fundamental principles. This article presents a progress report on an attempt to identify and develop a consensus on a set of candidate fundamental principles. A fundamental principle is less specific and more enduring than methodologies and techniques. It should be phrased to withstand the test of time. It should not contradict a more general engineering principle and should have some correspondence with "best practice". It should be precise enough to be capable of support and contradiction and should not conceal a tradeoff. It should also relate to one or more computer science or engineering concepts. The proposed candidate set consists of fundamental principles which were identified through two workshops, two Delphi studies and a web-based survey. © 2001 Elsevier Science Inc. All rights reserved.

1. Introduction

In the IEEE collection of standards, software engineering is defined as:

"(1) *The application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software, i.e. the application of engineering to software.* (2) *The study of approaches as in* (1)" (IEEE 610.12, 1991)

It is not easy to find, isolate and articulate the relevant principles that are fundamental to a discipline. Even in the more mature disciplines, where the fundamental principles are supposedly known, knowledge is often tacit. In the emerging discipline of software engineering, some attempts have been made, e.g. (Boehm, 1983; Davis, 1995), but a consensus has not yet developed.

In the 50-year history of software, various methodologies, methods and techniques have been proposed to facilitate the development of software responsive to needs. Most have proved to be more specific to the then-current state of technology than was understood at the time. As a result, many have subsequently been shown to be less universally applicable than originally intended. Despite a plethora of conferences and workshops over recent decades, and numerous periodicals, books and courses in the field, software engineering continues to lack universally recognized fundamental principles (McConnell, 1997, 1999).

The authors' interest in the identification of the fundamental principles of software engineering results from work in the development of software engineering practice standards. It is widely posited that practice standards should be based upon observation, recording and

* Corresponding author. Tel.: +1-514-396-8623; fax: +1-514-396-8684.
E-mail addresses: pbourque@ele.etsmtl.ca (P. Bourque), dupuis.robert@uqam.ca (R. Dupuis), aabran@ele.etsmtl.ca (A. Abran), james.w.moore@ieee.org (J.W. Moore), l.tripp@computer.org (L. Tripp), sibylle.wolff@sap.com (S. Wolff).
[1] Tel.: +1-514-987-3000x3479; fax: +1-514-987-8477.
[2] Tel.: +703-883-7396; fax: +703-883-5432.
[3] Tel.: +1-425-865-2732; fax: +1-425-865-6914.
[4] Tel.: +514-879-7250; fax: +514-879-7234.

consensual validation of implemented “best practices”. This strategy has resulted, though, in the development of a corpus of standards that are sometimes alleged to be isolated, unconnected and disintegrated, because each standard performs a local optimization of a single observed practice. It is hoped that the identification of a set of fundamental principles will provide a broad and rich framework for establishing relationships among groups of practice standards. A set of fundamental principles relating to the field could also help characterize the activities that differentiate software engineering from other computer-related activities and could help better define training programs. The identification of principles viewed as fundamental by the software engineering community would also provide a rich framework for analyzing and improving the Guide to the Software Engineering Body of Knowledge [5] (Abran et al., 2001). This guide is aimed at providing a topical access to the core subset of knowledge that characterizes the software engineering discipline.

In this paper, a progress report is presented on work carried out to identify and develop a consensus on a set of fundamental software engineering principles. [6] The paper begins with a discussion of the criteria for recognizing fundamental principles: what they are, their roles and how they relate to underlying concepts. In Section 3, the research methodology followed and the deliverables of each project phase are presented: two workshops, two Delphi studies and a web-based survey. The degree of consensus on the set of candidate principles and a brief discussion on each candidate are the subject of Section 4. Finally, a summary of the paper, and steps that could be undertaken to improve the set of candidate principles are presented in Section 5.

2. What are fundamental principles?

2.1. Underlying concepts versus fundamental principles

In discussing fundamental principles, there is sometimes confusion between such principles and what may be characterized as “underlying concepts”. Table 1 contrasts the two.

Table 1
Underlying concepts versus fundamental principles

Underlying concepts	Fundamental principles
Scientific	Engineering
Descriptive	Prescriptive
Validated through experiment	Validated through rigorous (but not necessarily experimental) assessment of practice
Judged on the basis of correctness	Judged on the basis of usability, relevance, significance, usefulness

Underlying concepts are to be regarded as scientific statements. They must be capable of validation by experiment and are judged on the basis of their correctness when subjected to experiment. By contrast, fundamental principles are to be regarded as engineering statements which prescribe constraints on solutions to problems or constraints on the process of developing solutions. They should be rigorously evaluated, but in practice rather than in the laboratory, and judged by whether or not they provide useful and substantial contributions to the successful solution of real problems of significant size and scope. In general, we would expect fundamental engineering principles to be strongly related to underlying scientific concepts.

An example of an underlying concept would be the theoretical results by Böhm and Jacopini (Böhm and Jacopini, 1966) indicating that essentially all programs can be expressed as combinations of statement sequences, branching and iterations, and that goto statements are not necessary. [7] In contrast, Dijkstra provided a corresponding prescriptive principle in his famous letter, “Go To Considered Harmful”, where he argues that programmers should actually apply the theoretical results in the construction of code (Dijkstra, 1968). [8]

2.2. Roles of fundamental principles

Fig. 1 illustrates the relationships sought among principles, standards and practices. It is believed that a body of fundamental principles for some branches of engineering has been recorded, e.g. (Vincenti, 1990) (most of the engineering branches have a history far longer than that of software engineering). Software engineering principles would, in the general case, be regarded as specializations of these principles. The software engineering principles would play the role of organizing, motivating, explaining and validating the practice standards. Implemented practices should be based on those practice standards.

[5] See www.swebok.org.

[6] For more detailed information on this research and notably the full set of participant comments, please see http://www.lrgl.uqam.ca/fpse.

[7] The candidate fundamental principles related to this underlying concept would at least be B, I and L. These candidate principles are presented in Table 3 and discussed in Section 4.

[8] Dijkstra’s principle, although useful as an example, is not one of those selected as fundamental in this study. First of all, it was not submitted by any of the Delphi participants. Secondly, one could also argue that Dijkstra’s principle does not satisfy one of the criteria – independence from specific methodology, in this case, design by functional decomposition.

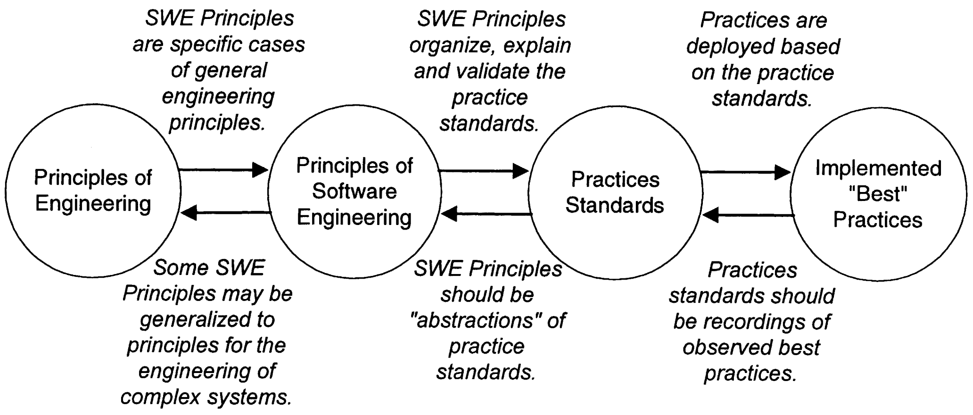
SWE Principles are specific cases of general engineering principles.
SWE Principles organize, explain and validate the practice standards.
Practices are deployed based on the practice standards.
Principles of Engineering
Principles of Software Engineering
Practices Standards
Implemented "Best" Practices
Some SWE Principles may be generalized to principles for the engineering of complex systems.
SWE Principles should be "abstractions" of practice standards.
Practices standards should be recordings of observed best practices.

Fig. 1. Relationship between principles and practice.

Working from the specific toward the general, practice standards would be recordings and idealizations of observed and validated "best practices". The software engineering principles would be abstractions of the practice standards. Furthermore, software engineering principles might be candidates for generalization to the status of general engineering principles, particularly when complexity is a concern.

2.3. Criteria for the recognition of fundamental principles

The following criteria (possibly better regarded as heuristics or meta-principles) were developed for the recognition of fundamental principles:

- Fundamental principles are less specific than methodologies and techniques, i.e. specific methodologies and techniques may be selected, within a particular technological context, to accomplish the intent of fundamental principles.
- Fundamental principles are more enduring than methodologies and techniques, i.e. fundamental principles should be phrased in a way that will stand the test of time, rather than in the context of current technology.
- Fundamental principles are typically discovered or abstracted from practice and should have some correspondence with best practices.
- Software engineering fundamental principles should not contradict more general fundamental principles.
- A fundamental principle should not conceal a tradeoff. By that we mean that a fundamental principle should not attempt to prioritize or select from among various qualities of a solution; the engineering process should do that. Fundamental principles should identify or explain the importance of the various qualities among which the engineering process will make trades.
- ... but, there may be tradeoffs in the application of fundamental principles.
- A fundamental principle should be precise enough to be capable of support or contradiction.
- A fundamental principle should relate to one or more underlying concepts.

3. Research methodology and project deliverables by phase

The project was prompted by a 1996 decision of the IEEE Software Engineering Standards Executive Committee to begin efforts to identify a list of fundamental principles for software engineering. The research methodology that has been followed to identify a candidate list is summarized in Fig. 2. The deliverables of each project phase are described next, from the project kickoff meeting in 1996 to the larger web-based survey in 1999.

3.1. 1996 kickoff workshop: process and deliverables

Three days of discussion at the kickoff workshop of the Software Engineering Standards Symposium (SES'96) held in Montreal resulted in (Jabir and Moore, 1998):

- Observations on the nature of fundamental principles;
- Criteria for identifying and evaluating candidate principles;
- Examples and counter-examples of principles;
- A recommendation that a Delphi study be organized and conducted for identifying an initial set of candidate fundamental principles. This Delphi study would be conducted among a group of software engineering experts and would use the criteria developed in Montreal for identifying and evaluating these candidate fundamental principles. It was also recommended that a subsequent workshop be held to analyze the results of this Delphi study.

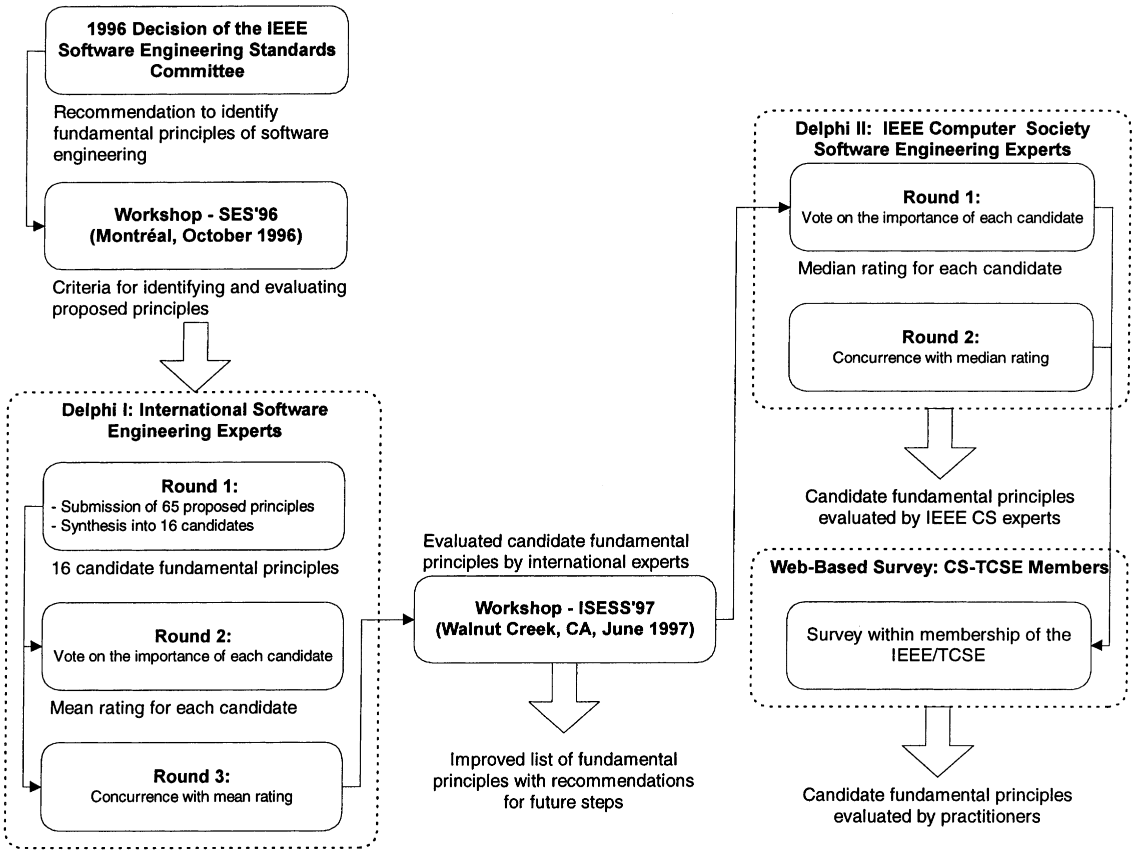
1996 Decision of the IEEE Software Engineering Standards Committee
Recommendation to identify fundamental principles of software engineering
Workshop - SES'96 (Montréal, October 1996)
Criteria for identifying and evaluating proposed principles
Delphi I: International Software Engineering Experts
Round 1:
- Submission of 65 proposed principles
- Synthesis into 16 candidates
16 candidate fundamental principles
Round 2:
Vote on the importance of each candidate
Mean rating for each candidate
Round 3:
Concurrence with mean rating
Evaluated candidate fundamental principles by international experts
Workshop - ISESS'97 (Walnut Creek, CA, June 1997)
Improved list of fundamental principles with recommendations for future steps
Delphi II: IEEE Computer Society Software Engineering Experts
Round 1:
Vote on the importance of each candidate
Median rating for each candidate
Round 2:
Concurrence with median rating
Candidate fundamental principles evaluated by IEEE CS experts
Web-Based Survey: CS-TCSE Members
Survey within membership of the IEEE/TCSE
Candidate fundamental principles evaluated by practitioners

Fig. 2. Overview of project steps.

3.2. Delphi 1: process and deliverables

The objective of this first Delphi study conducted by email in the spring of 1997 was to identify an initial set of candidate fundamental principles. The desire to consult eminent representatives of the international community went hand in hand with the selection of the Delphi method. This technique consists in forming a panel of experts, who are not told each other's identities until the end of the exercise so that the prestige or power of one member of the group does not unduly influence the course of the discussion.

In Round 1, the international experts were asked to submit proposals, based on the criteria already established in the kickoff workshop. Each one was asked to draw up a list of the five fundamental principles they felt were most pertinent. They were also invited to add any amount of commentary or explanations so that the Delphi study coordinators could better understand and explain their selection to the other participants of Round 2. This message was sent to 52 international software engineering experts. The list of contacted experts contacted was selected by the authors of this paper and by informally consulting some of the authors' colleagues. Thirteen individuals responded, which means that 65 proposed principles were obtained and are listed in Appendix A.

Then, two Delphi study coordinators consolidated these 65 suggestions into a smaller number of principles which met the criteria of the SES'96 workshop. This produced the 16 candidate fundamental principles listed in Appendix B. At this stage, the objective was to include the largest number of suggestions possible, while at the same time trying to reduce overlap among the candidates.

In Round 2, participants were asked to rate each of the 16 candidate fundamental principles from Round 1 on a scale of 1 to 10. Brief explanations, that were essentially culled from the Round 1 submissions, were provided for each candidate principle. Participants were also asked to comment on their ratings, which most of them did. This also provided them with the opportunity to contest the way in which their initial five suggestions had been consolidated into the 16 candidate fundamental principles.

The goal of Round 3 of a Delphi study is to reinforce and confirm the ratings that emerge. Therefore, participants were sent the mean scores of each candidate fundamental principle. They were then asked whether or not they agreed with the rating and to add more comments if need be.

Table 2
Delphi 1 study – international experts

Participant	Organization as of Delphi 1	Country as of Delphi 1
M. Azuma	Waseda University	Japan
F.P. Brooks	University of North Carolina	USA
R.N. Charette	ITHABI Corp.	USA
P. DeGrace	Consultant	USA
C. Ghezzi	Politecnico di Milano	Italy
T. Gilb	Result Planning Ltd.	Norway
B. Littlewood	City University	United Kingdom
S. MacDonell	University of Otago	New Zealand
T. Matsubara	Matsubara Consulting	Japan
J. Musa	Consultant	USA
R.S. Pressman	R.S. Pressman&Associates	USA
M. Shaw	Carnegie-Mellon University	USA

The value of this initial candidate list lies primarily in the expertise of the respondents. Table 2 lists the participants (12 of the 14) [9] who agreed to have their names published. [10] The other two participants chose to remain anonymous. We believe that the list represents a group that is diverse in terms of nationality, approach to software engineering, and theoretical versus practical experience.

It must be strongly emphasized that the output of a Delphi study does not represent the single opinion of each individual participant, but rather a group view of the topics being investigated, and that it documents the degree of consensus (or lack of it) on such a group view.

3.3. ISESS'97 workshop

This initial Delphi study was followed by a workshop at the IEEE-International Software Engineering Standards Symposium – ISESS'97, [11] where some twenty participants discussed the findings. Based on this review and the lessons learned, this second workshop produced a list of improved criteria, as well as a more refined list of fundamental principles, [12] as illustrated in Table 3. These improvements were to be incorporated in the next Delphi study and in the web-based survey.

[9] One participant did not participate in Round 1 of Delphi 1 and one participant did not participate in Round 3.

[10] For a short biography of each participant, consult the following address: http://www.lrgl.uqam.ca/fpse/emailfirstdelphi.pdf.

[11] ISESS'97, Third International Symposium and Forum on Software Engineering Standards, Walnut Creek, CA, IEEE Computer Society, June 1997.

[12] Based on the mean scores of each candidate in Delphi 1, on the level of consensus among the experts and on the expert comments, the workshop participants concluded to make the following changes to the list of candidate principles. Candidate C in Table 3, discussed at the SES'96 workshop, was adopted instead of candidate 3 in Appendix B. Candidate N, also discussed at the SES'96 workshop, was added to the list. Candidates 14 and 15 were dropped from the list.

3.4. Delphi 2 study: process and deliverables

The first consultation, although very fruitful, had obvious limitations, notably the limited number of participants and the fact that their suggestions could have been consolidated differently, producing a different list of principles altogether. Consequently, a second Delphi study was carried out, this time among a group of software engineering experts drawn from a different pool. These individuals were deemed to be experts by virtue of the fact that they held an "official" software engineering position within the IEEE Computer Society.

The contacted experts were members of the Executive Committee of the *Technical Council on Software Engineering* or members of the editorial committees of IEEE *Software* or IEEE *Transactions on Software Engineering*. Thirty-one experts of the 72 contacted agreed to participate in this two-round Delphi study.

Table 3
List of candidate fundamental principles (in alphabetical order)

A.	Apply and use quantitative measurements in decision-making
B.	Build *with* and *for* reuse
C.	Control complexity with multiple perspectives and multiple levels of abstraction
D.	Define software artifacts rigorously
E.	Establish a software process that provides flexibility
F.	Implement a disciplined approach and improve it continuously
G.	Invest in the understanding of the problem
H.	Manage quality throughout the life cycle as formally as possible
I.	Minimize software component interaction
J.	Produce software in a stepwise fashion
K.	Set quality objectives for each deliverable product
L.	Since change is inherent to software, plan for it and manage it
M.	Since tradeoffs are inherent to software engineering, make them explicit and document them
N.	To improve design, study previous solutions to similar problems
O.	Uncertainty is unavoidable in software engineering. Identify and manage it

Table 4
Delphi 2 – list of participants

Maarten Boasson	Richard Kemmerer	Shari Lawrence Pfleeger
Shawn Bohner	Barbara Kitchenham	Vaclav Rajlich
Terry Bollinger	Reino Kurki-Suonio	Rami R. Razouk
Andy Bytheway	David John Leciston	Sam Redwine
Carl Chang	Keith Marzullo	Mary Lou Soffa
James Cross II	Nancy Mead	David S. Wile
Peter Eirich	Stephen J. Mellor	Linda Wills
Bill Everett	Ware Myers	James Withey
Gene F. Hoffnagle	Michael Olsem	
Mehdi Jazayeri	Linda Ott	

Participants in the second Delphi study who agreed to have their names published are listed in Table 4. One participant chose to remain anonymous, one withdrew and one did not respond during Round 2.

3.5. Web-based IEEE-CS TCSE survey

The output of the second Delphi study was to serve as an input to a web-based survey of the members of the Technical Council on Software Engineering (TCSE) of the IEEE-Computer Society (IEEE-CS), with the cooperation of the IEEE Computer Society.

A survey instrument was prepared and pre-tested with a limited sample of 30 participants. Required adjustments to questions were made based on feedback received. Survey participants were asked to rate each candidate fundamental principle and were also given the opportunity to provide additional comments at the end of the survey questionnaire.

An introductory letter was prepared by the 1999 IEEE-CS president, Mr. Leonard Tripp, and sent by email to all members of the IEEE-TCSE. A total of 3509 TCSE members with valid email addresses were contacted. Out of these members, 574 answered the web-based survey, representing approximately 16% of the targeted population.

3.5.1. Demographics of the respondents

- International participation: of the 556 respondents who indicated their country of residence, 50% were from the US, while the other 50% were from 48 countries, 11 countries having over 10 respondents each.
- Educational background: respondents were from a significant mix of educational backgrounds, ranging from computer science only, to engineering only, to math only, but mostly from any combination of these.
- Highest degree: 43% of the respondents indicated that they had a Ph.D., and 35% indicated a Master's degree.
- Years in software engineering: 36% of the respondents indicated that they had over 20 years of experience in software engineering, while another 37% indicated that they had between 10 and 20 years of experience.
- Years of practice in industry: 23% of the respondents indicated that they had over 20 years of experience in industry, and 36% between 10 and 20 years.
- Employer's line of business: the major categories of line of business of employers were: R&D (30%) and software (23%), and the balance from a variety of businesses, with only 6% from academia.
- Type of software: the larger portion of respondents by type of software were from the MIS domain with 38%, followed by real-time software (29%) and scientific software (14%).

4. Discussion and degree of consensus on the candidate set of fundamental principles

Table 5 summarizes the degree of consensus on the candidate fundamental principles from each group of participants. Subsequently, the candidate principles are discussed separately based both on the analysis of Table 5 and on the rich set of comments provided by the study participants. A summary of supportive comments is presented for each candidate principle, followed by some issues which were identified by the study participants.

For the first Delphi study with the group of 12 experts, the aggregate ratings of this group of experts are represented by the mean score (1 is low, and 10 is high), while in the second column the number of yes votes indicates the number of experts (out of 12) who rallied to the mean score. From the second Delphi study on, it became apparent that the median was more appropriate for this type of study, and the methodology was modified accordingly. The third column represents the median score, therefore, and the fourth column the number of yes votes out of 29 [13] participants. Finally, the fifth column indicates the median score for the 574 web-based participants, while the sixth column provides the standard deviation for this larger sample.

4.1. Apply and use quantitative measurements in decision-making (A)

To the surprise of the authors, the level of support for this candidate principle was quite weak, especially among the participants of Delphi 1 and 2. Essentially, two points of view prevailed. Some experts argued that measurement is fundamental to engineering in general and without it there is no engineering. For them, measurement should always be applied and should be qualified if needs be. Measurement is key to driving decisions and progress in the field. Others believe that

[13] 29 out of the 31 second Delphi study participants took part in Round 2.

Table 5
Overview of participant consensus – documented at project phases

		Delphi 1		Delphi 2		Survey	
		Mean score	Yes votes	Median score	Yes votes	Median score	S.D.
A.	Apply and use quantitative measurements in decision-making	7.6	7/12	7	13/29	8	2.4
B.	Build with and for reuse	8	7/12	9	17/29	8	2.3
C.	Control complexity with multiple perspectives and multiple levels of abstraction	N/A	N/A	8	23/29	8	2.4
D.	Define software artifacts rigorously	6.4	6/12	8	22/29	8	2.5
E.	Establish a software process that provides flexibility	7.6	7/12	8	21/29	8	2.3
F.	Implement a disciplined approach and improve it continuously	6.9	4/12	8	19/29	9	2.4
G.	Invest in the understanding of the problem	8.7	7/12	10	29/29	10	2
H.	Manage quality throughout the lifecycle as formally as possible	7.8	7/12	9	20/29	8	2.5
I.	Minimize software component interaction	7.3	8/12	9	25/29	7	2.7
J.	Produce software in a stepwise fashion	7.7	7/12	8	23/29	7	2.7
K.	Set quality objectives for each deliverable product	7.7	8/12	8	20/29	8	2.3
L.	Since change is inherent to software, plan for it and manage it	9.1	9/12	10	26/29	9	2
M.	Since tradeoffs are inherent to software engineering, make them explicit and document them	8.4	8/12	9	25/29	9	2.3
N.	To improve design, study previous solutions to similar problems	N/A	N/A	9	24/29	8	2.1
O.	Uncertainty is unavoidable in software engineering. Identify and manage it	8	8/12	10	25/29	8	2.5

this principle has too many caveats for it to be universally applicable or even universally desirable. In their opinion, for example, the principle is not always applicable due to excessive cost or to the low level of maturity of software engineering. Some participants expressed the view that one can focus too strongly on measurement to the detriment of better judgment, and that measurement constitutes only one form of input to the decision-making process.

4.2. Build with and for reuse (B)

Even though the scores on this principle are quite high; Table 5 shows that the level of consensus on these scores is weaker. Most of the participant comments on this principle revolved around the idea that though reuse is desirable and important, it is not always pertinent. It depends on many factors, notably the economic context and the delivery schedule. Some participants noted that the risk associated with the level of reliability of reused artifacts must always also be considered. However, it was also asserted that this principle does not go far enough: the software engineering field should be able to produce a set of descriptions on how to reuse, and a set of artifacts at many different levels of abstraction which can be reused, as well as a set of catalogues which make it easy to navigate throughout all of this.

4.3. Control complexity with multiple perspectives and multiple levels of abstraction (C)

The participants in the ISESS'97 workshop adopted this candidate principle as a replacement for candidate 3 of Delphi 1 (see Appendix B) which was withdrawn due to a low mean score and a low level of consensus on this mean score (mean score: 4.9/10, number of yes votes: 6/12). Candidate C was discussed at length at the SES'96 workshop.

Table 5 shows that the median score and the level of consensus on this median score for candidate C are both substantially higher than for the previous candidate. Even so, there were some participants who emphasized that this was not the only way to reduce complexity, and in fact some stated that multiple perspectives and levels of abstraction could indeed add complexity. For example, one participant stated that you could simplify the requirements in order to control complexity.

4.4. Define software artifacts rigorously (D)

Table 5 shows that the participants of Delphi 2 and of the web-based survey were stronger supporters of this candidate principle than the Delphi 1 participants, though it is difficult to understand why. Supporters of this candidate principle indicated that rigor was essential to conducting software engineering activities. Others expressed concern over what is meant by rigor. Does it imply being formal, or, for example, the application of standards? There were some participants who emphasized the fact that the desired level of rigor is dependent upon the situation and, notably, the level of criticality of the software.

4.5. Establish a software process that provides flexibility (E)

Table 5 shows that there is strong support for this candidate principle, which is closely related to candidate F. Supporters of this candidate principle are of the opinion that flexibility in the process is required since most software engineering projects must deal, notably, with accelerated delivery schedules, as well as changing and incomplete requirements. Others articulated the need for a variety of processes to be defined and used for projects of, for example, various degrees of scopes. In contrast, some participants expressed concern that in reality too much flexibility can often lead to having no defined process being followed at all.

4.6. Implement a disciplined approach and improve it continuously (F)

Table 5 shows that the level of support for this candidate principle was strongest among the web-based survey participants. One participant believes that this is part of the "essence" of science and engineering in general. A second participant stated that "once a project is staffed, the quality of the processes used to develop software will largely determine the quality, timeliness, and cost effectiveness of the result". However, other participants took the position that the disciplined approach or process itself could become the focus of attention at the expense of the product to be developed, and that a "disciplined approach" is too often viewed as a "silver bullet". Interestingly, very few participants commented on the continuous improvement portion of this candidate principle.

4.7. Invest in the understanding of the problem (G)

This candidate principle is the highest rated candidate in the proposed list. In fact, all 29 participants in Delphi 2 rallied to the median score of 10. This is also the median score of the web-based survey. As one participant eloquently said: "This is the absolute pre-requisite to everything else." A second participant emphasized that investment was the issue at hand and not cost. However, a few participants voiced the opinion that this candidate was most certainly not specific to software engineering and that it was impossible to disagree with it.

4.8. Manage quality throughout the life cycle as formally as possible (H)

Table 5 shows that even though the level of support for this candidate principle is quite strong, the level of consensus around the scores is not as high, as evidenced by the number of yes votes in Delphi 1 and Delphi 2. Many comments regarding this candidate principle were formulated as questions. What does quality precisely mean? Is the definition of quality dependent on the context? Is it dependent on the stakeholder? How do you measure "quality", and who measures it? Could or should "as formally" be replaced with "as explicitly"? Could "as possible" be substituted with "as practical"? Could the principle be simply stated simply as "Manage quality throughout the life cycle"? Additionally, how does this candidate principle interact with candidate principle K?

4.9. Minimize software component interaction (I)

For some reason, the Delphi 2 participants scored this candidate principle higher than the Delphi 1 and the web-based survey participants. Additionally, the level of consensus on the median score of Delphi 2 participants is also higher. While some participants believed that this candidate principle reflected something "deeper" regarding the locality of processing and was essential to the control of complexity, others put forward some very pertinent questions, such as, for example: Should "minimize" be replaced by "manage" or "precisely specify"? Should "interaction" be restated as "interdependencies"? Does this candidate principle apply to all types of software? If this candidate principle refers to the coupling issue, should there be a complimentary candidate principle referring to the cohesion issue? Can this candidate principle be over-applied, leading to poorly or non-integrated software?

4.10. Produce software in a stepwise fashion (J)

Table 5 shows that the level of support and the level of consensus on the scores for this candidate principle were quite consistent across the three studies. This level of support is well illustrated by one participant's comment that stated that this candidate principle was the "most fundamental" insight that we have arrived at over the past 20 years. However, this same participant, and others as well, emphasized that the application of this candidate principle is not applicable to all categories of software and problems, nor to all phases of the software life cycle. Indeed, one participant went even further, by stating that, if we were able to produce software in a single step, we should. This participant believes that this is not a principle, but a consequence of our current inability to do better.

4.11. Set quality objectives for each deliverable product (K)

As with candidate J, the level of support and the level of consensus around the scores for this candidate principle are quite consistent across the three studies. As put forward by some participants, the level of quality, independently of how one defines quality, does not need to be the same for all categories and for all components of a given software. Others questioned what quality precisely means in this candidate principle and how this candidate principle relates to candidates A and H.

4.12. Since change is inherent to software, plan for it and manage it (L)

This principle is very highly rated, with a strong level of consensus, by the participants in all three studies. One participant identified many ways to apply it. For example, portions of software which change frequently should be isolated from portions which change less. Software should be self-descriptive and programmed with widely used languages. A maintenance plan should be incorporated into the project plan. Review points should be incorporated into a project plan which provides for changes in requirements as well as for changes in response to work in progress. Software components which use vendor-specific capabilities or technology that is likely to change should be better documented. A few participants emphasized that not all software changes frequently, and not all changes can be foreseen.

This principle provides a clear example of the criterion that "A fundamental principle should be precise enough to be capable of support or contradiction". If this study had been performed 20 years ago, we would probably have obtained some variant of "The requirements must be firm and fixed" (candidate 14 of Delphi 1). Candidate 14 was not retained after Delphi 1 due to the weak support for this candidate. Today, most software engineers understand that freezing is an undesirable action because it means commitments to a set of requirements which are obsolete upon delivery of the system. This example also illustrates the concept that a principle is judged on the basis of utility.

4.13. Since tradeoffs are inherent to software engineering, make them explicit and document them (M)

Table 5 shows that the level of support is consistently strong for this candidate principle. As stated by one participant expressed it, you cannot meet an arbitrarily large number of good goals. Deliberate choices based on sound analysis of these choices must be made. Others noted that a proper understanding of the problem as identified in candidate principle G is essential to making these tradeoffs, as are the measurements identified in candidate principle A. It was also noted that the notion of constraint should not be confused with that of tradeoff. Tradeoffs should be viewed as one mechanism for dealing with constraints.

4.14. To improve design, study previous solutions to similar problems (N)

The SES'96 workshop participants proposed this candidate principle as an example of a fundamental principle. However, as with principle C, it did not emerge from the consolidation of the proposals made by the Delphi 1 participants. The participants of the ISESS'97 workshop viewed this as an important omission and added it to the list. It received strong support in the two subsequent studies. One participant noted that one should not restrict oneself to studying only previous software solutions to similar problems. Others inquired as to why this candidate principle was limited to design and as to how to identify which characteristics to use in determining similarity.

4.15. Uncertainty is unavoidable in software engineering. Identify and manage it (O)

Interestingly again, this candidate received stronger support from the participants of Delphi 2 than from the participants of the other two studies. One participant postulated that there was probably more uncertainty in software engineering than in other engineering disciplines because it relies so much on human expertise. Other participants questioned what precisely is meant by uncertainty. Does it mean risk; changing business needs; uncertain completion schedules; uncertain staffing; uncertain markets for the software; all of the above; or something else? There were some participants who also inquired as to how this candidate principle interacts with the notions of measurement and decision-making put forth in candidate A and the notion of change identified in candidate L.

5. Summary and next steps

This paper has reported on a series of efforts undertaken to try to identify a set of fundamental principles of software engineering. A first workshop was held at the Forum on Software Engineering Standards Issues of 1996 (SES'96) to establish what a fundamental principle is and what the criteria are to which it should conform.

A Delphi study was then conducted in 1997 over the Internet among 14 international software engineering experts to identify a first candidate list of fundamental principles of software engineering. A second workshop was held at the International Symposium on Software

Engineering Standards of 1997 (ISESS'97) to eliminate or reformulate some of the principles and the criteria. Subsequently, a second Delphi study was conducted in 1998 among 31 software engineering experts holding software engineering positions within the IEEE Computer Society. From these studies, a list of fifteen candidate fundamental principles of software engineering was compiled. Finally, an electronic survey was conducted among the membership of the Technical Council on Software Engineering to confirm the relevance of these candidate principles for practitioners and to help determine which of these 15 candidate principles are indeed fundamental.

Based on the participant feedback and the many pertinent issues that were brought up, a workshop is now planned, possibly followed by one or more consensus formation activities (such as Delphi studies) to develop paragraph-length explanations for each principle. The object of these paragraph-length explanations would be to provide better guidance on how each principle should be interpreted.

The current set of fundamental principles was developed based on domain experts' opinions, in successively larger samples. While highly valuable in developing and documenting the level of consensus on the process output in terms of speed and cost, this type of research has inherent methodological limitations which must be addressed in the future.

Techniques other than opinion surveys should now also be investigated, through empirical designs for corroborating these principles both within current theories proposed in the field of software engineering and with observation of their implementation in currently recommended best practices.

Other activities can be planned as well: analysis of the body of current standards, of generally accepted knowledge in software engineering as described in the Guide to the Software Engineering Body of Knowledge, and of university programs in software engineering to assess the extent to which fundamental principles are covered.

As shown in Fig. 1, the interaction between the candidate fundamental principles proposed in this paper for software engineering and the more generic principles of engineering, notably, is an issue that must also be investigated. However, as indicated in Section 1, fundamental principles are often tacit in the more mature engineering disciplines. A structured comparison of the work by Vincenti (Vincenti, 1990) with the list of candidate principles proposed in this paper is seen as an interesting avenue in addressing this issue. Vincenti proposes a taxonomy of engineering knowledge in general based on the historical analysis of five case studies in aeronautical engineering covering a roughly 50-year period. Maibaum adopted Vincenti's categories of engineering knowledge to identify issues which need be addressed to increase the maturity of software engineering practice (Maibaum, 2000).

A thorough and structured analysis of the participant comments would provide very valuable research material for further exploration. An example of such an analysis on the role of measurement in the set of candidate fundamental principles can be found in (Wolff, 1999).

Through a judicious combination of these proposed next steps, it is hoped that better guidance will be available on how to interpret the candidate principles, that the potential flaws of the opinion-based studies presented in this paper will be pinpointed and that the set of candidate principles can in due course be judged on the basis of usability, relevance, significance and usefulness.

Acknowledgements

We would like to thank everyone who contributed their ideas to this work: participants of Delphi studies, workshops and web-based survey, and the anonymous reviewers of this paper. The authors would also like to thank the IEEE Computer Society and especially John Keaton for their continuing support.

Appendix A. Initial 65 proposed principles submitted in Round 1 of the Delphi 1 study (in alphabetical order)

1. Accept change.
2. Accept uncertainty in estimates.
3. Adopt a managed improvement-oriented 'production' cycle.
4. Anticipation of change.
5. Build a model of the solution that can be assessed for quality before any code is generated.
6. Build a transparent architecture.
7. Build with and for reuse.
8. Code in widely used, high-level languages with an expectation of future maintenance, and the need to revise/update and correct software over time. Identify an expected lifetime for the software (including worst case).
9. Controlling risk. There are large quantities of uncertainty, and risk of deviation from plans, in any project. You cannot eliminate risk, but you can document it, plan and design for it, accept it, measure it, and reduce it to acceptable levels.
10. Distinguish essential properties from incidental properties. Enforce the essential properties and make provisions for the incidental properties to be modified easily.
11. Do not displace objectives with means.

12. Document only what is necessary.
13. Document structure, operations, and limitations. Isolate code that is likely to change: where work is based on vendor specific capabilities, where technology is likely to change, and/or where user requirements are likely to change.
14. Establish a software process that provides flexibility but still encourages discipline.
15. Establish a testing strategy that incorporates systematic testing methods.
16. Every project should plan and quantitatively specify its reliability strategies (the blend of fault prevention, fault removal, and fault tolerance) based on its reliability objective(s).
17. Every software project should develop an operational profile or profiles in consultation with users or their representatives, so that expected usage can be quantified and development and testing focus placed on the most used and/or most critical functions.
18. Every software project should set a reliability objective or objectives for its deliverable product(s). Actual reliability achieved should be estimated at various points in the project and compared with the objective(s), the ratio being used to manage the project.
19. Explore multiple design alternatives, evaluate them objectively, and make deliberate choices among them.
20. Generality.
21. Goal of software engineering (management) is improving quality, reducing costs (Productivity improvement) and delivering in time. But continuous efforts for improving the environment, motivating and educating people (engineering staff), and improving process are more important.
22. Goals beat all. Meeting requirements is more fundamental than any other process or principle.
23. Identify and profile (see IEEE Std. 1003.0) the environment for operations; do this in terms of standards where applicable.
24. Implement mechanisms to reduce entropy of processes and products.
25. Incorporate a backup and maintenance plan into the project plan. Make sure you can access the source code and documentation at the end of the lifecycle as well as at the beginning.
26. Incrementality.
27. Make careful trades between generality and power.
28. Manage software quality upstream. (Upper stream is more important.)
29. Measure what matters, not what is handy. Predict what to expect from the measurements before taking them, and use the results to guide adjustments.
30. Minimize software component interaction.
31. Once a project is staffed, the quality of the processes used to develop software will largely determine the quality, timeliness, and cost effectiveness of the result.
32. Produce independent software.
33. Produce consistent software.
34. Produce modular software.
35. Produce self-descriptive software.
36. Produce simple software.
37. Projects should choose their tools and methodologies based on quantitative criteria, measuring their effect on reliability, schedule, and cost with respect to cost of implementation.
38. Quality (the balance of reliability, schedule, and cost with respect to user needs) must be defined specifically for each product.
39. Quality Requirements must be specified for all quality characteristics, and quality must be measured for all requirements.
40. Quality should be into the product and the process.
41. Quantification is mandatory for control. The multiple concurrent quality and cost demands of most systems, means that a quantified and testable set of requirements is necessary, to get control over quality and costs.
42. Reasonable balance. Requirements must be balanced; you cannot meet an arbitrarily large number of arbitrarily good goals.
43. Recognize risk.
44. Rigor (which can lead to formality).
45. Separate concerns. Isolate changing from minimum-changing parts of the software/system.
46. Separation of concerns.
47. Software engineering is *not* engineering.
48. Software quality evaluation must be scientific and quantitative. But it must be practical and cost effective.
49. Software really is different (from conventionally engineered products).
50. Strive for predictable patterns for data, program architecture, interfaces, and procedural logic, and whenever possible, work to reuse existing patterns.
51. Test assumptions.
52. The most important determinants of the quality, cost, and timeliness of software products are the capability and motivation of the engineer or engineers who produce them.
53. The requirements for software products are inherently ill defined.
54. The tools, methods, and support systems must be designed and selected to support the software engineers and the processes they use.
55. There are quite stringent limits to the dependability that can be achieved and demonstrated for software-based systems.

56. To build quality software products, the requirements must be firm and fixed.
57. Understand current requirements, and incorporate into the project plan review points that provide for changes in requirements as well as changes in response to work in progress.
58. Understand dependency mechanism causing rework.
59. Understand your problem. No amount of rote application of analysis, process, method, formalism, or tools can substitute for genuinely understanding what you are doing.
60. Understanding the application domain is the most important part of software engineering.
61. Unnecessary difficulty, novelty and complexity are the main enemies of the software engineer.
62. Use visual presentations for better quality management.
63. View requirements from multiple perspectives.
64. Work with the customer to gain a clear understanding of the problem and an indication of what is not yet understood.
65. You cannot have it all. Some goals will be deemed by you, and by your customer, to have higher priority than others, at particular times, places and conditions.

Appendix B. Candidate principles that were consolidated from the 65 proposals found in Appendix A (in alphabetical order)

1. Apply and use quantitative measurements in decision-making.
2. Build with and for reuse.
3. Deal with different individual aspects of the problem by concentrating on each one separately.
4. Define software artifacts rigorously.
5. Establish a software process that provides flexibility.
6. Implement a disciplined process and improve it continuously.
7. Invest in the understanding of the problem.
8. Manage quality throughout the life cycle as formally as possible.
9. Minimize software components interaction.
10. Produce software in a step-wise fashion.
11. Set quality objectives for each deliverable product.
12. Since change is inherent to software, plan for it and manage it.
13. Since tradeoffs are inherent to software engineering, make them explicit and document them.
14. The requirements must be firm and fixed.
15. The tools, methods, and support systems must be designed and selected to support the software engineers.
16. Uncertainty is unavoidable in software engineering. Identify and manage it.

References

Abran, A., Moore, J.W. (Exec. Eds), Bourque, P., Dupuis R. (Eds.), 2001. Guide to the Software Engineering Body of Knowledge – Trial Version (Version 0,95). Montréal, IEEE Computer Society (can be downloaded from www.swebok.org).

Boehm, B.W., 1983. Seven basic principles of software engineering. The Journal of Systems and Software 3 (1), 3–24.

Böhm, C., Jacopini, G., 1966. Flow diagrams, turing machines and languages with only two formation rules. Communications of the ACM 9 (5), 366–371.

Davis, A.M., 1995. 201 Principles of Software Development. McGraw-Hill, New York.

Dijkstra, E.W., 1968. Go to statement considered harmful. Communications of the ACM 11 (3), 147–148.

IEEE Std 610.12, 1991. IEEE Standard Glossary of Software Engineering Terminology. Corrected Edition, February.

Jabir, Moore, J.W., 1998. A Search For Fundamental Principles of Software Engineering. Report of a Workshop Conducted at the Forum on Software Engineering Standards Issues, published in Computer Standards&Interfaces – The International Journal on the Development and Application of Standards for Computers, 1998. Data Communications and Interfaces. North-Holland, Elsevier, Amsterdam, 19(2), 155–160 (the participants at this workshop dubbed their group, "Jabir").

Maibaum, T.S.E., 2000. Mathematical foundations of software engineering. In: Proceedings of the Conference on the Future of Software Engineering, ACM, pp. 161–172.

McConnell, S., 1997. Software's ten essentials. IEEE Software 14 (2), 143–144.

McConnell, S., 1999. Software engineering principles. IEEE Software 16 (2), 6–8.

Vincenti, W.G., 1990. What Engineers Know and How They Know It – Analytical Studies from Aeronautical History. Johns Hopkins, Baltimore, MD and London.

Wolff, S., 1999. La place de la mesure au sein des principes fondamentaux du génie logiciel. Masters Thesis. Université du Québec à Montréal, Montréal, Québec, Canada.