JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 2001; **13**:3–30

Research

Types of software evolution and software maintenance

Ned Chapin[1,*,†], Joanne E. Hale[2], Khaled Md. Khan[3], Juan F. Ramil[4] and Wui-Gee Tan[5]

[1]*InfoSci Inc., Menlo Park CA 94026–7117, U.S.A.*
[2]*Department of Information Systems, Statistics, and Management Science, University of Alabama, Tuscaloosa AL 35487–0226, U.S.A.*
[3]*School of Computing and Information Technology, University of Western Sydney, Kingswood, NSW 2747, Australia*
[4]*Department of Computing, Imperial College, London SW7 2BZ, U.K.*
[5]*Institute of System Science, National University of Singapore, Singapore 117611, Singapore*

SUMMARY

The past two decades have seen increasing sophistication in software work. Now and in the future, the work of both practitioners and researchers would be helped by a more objective and finer granularity recognition of types of software evolution and software maintenance activities as actually done. To these ends, this paper proposes a clarifying redefinition of the types of software evolution and software maintenance. The paper bases the proposed classification not on people's intentions but upon objective evidence of maintainers' activities ascertainable from observation of activities and artifacts, and/or a before and after comparison of the software documentation. The classification includes taking into account in a semi-hierarchical manner evidence of the change or lack thereof in: (1) the software, (2) the documentation, (3) the properties of the software, and (4) the customer-experienced functionality. A comparison is made with other classifications and typologies. The paper provides a classified list of maintenance activities and a condensed decision tree as a summary guide to the proposed evidence-based classification of the types of software evolution and software maintenance. Copyright © 2001 John Wiley & Sons, Ltd.

KEY WORDS: software evolution management; software maintenance management; maintainer activities; maintenance terminology; evolution terminology; software support; empirical studies

1. INTRODUCTION

Our motivations for proposing a finer grained objective classification of the types of activities involved in software evolution and software maintenance have been primarily the following three:

*Correspondence to: Ned Chapin, InfoSci Inc., Box 7117, Menlo Park, CA 94026–7117, U.S.A.
†E-mail: NedChapin@acm.org

Received 11 July 2000
Revised 25 November 2000

- Practitioners undertake a great variety of activities in accomplishing software evolution and software maintenance, but such variety has often been obscured by the use of broadly inclusive terms such as 'perfective maintenance'. As has been long reported in the field, practitioners use diverse skill sets and operate in a wide variety of situations [1–3]. Practitioners could more easily receive recognition for the results they achieve and the effort, skill, and knowledge they apply if an objective evidence-based finer-grained classification were used for software evolution and software maintenance work.
- Managers involved in software evolution and software maintenance have great difficulty in defining, justifying, budgeting, staffing, supervising, accounting for, marshalling resources for, and assessing the work when coarse non-objective measures are used [4]. Things not appropriately named are often undervalued and treated as miscellaneous or trivial, since communication about them is made more cumbersome.
- Researchers need relevant descriptive names for the types of activities they observe when studying software evolution and software maintenance. As has been noted elsewhere, researchers have been using the same terminology but with different meanings, and using different terminology but with the same meanings, for common activities [5]. This complicates researchers' work when trying to build on the theoretical and empirical work of other researchers in software evolution and software maintenance.

Our main objectives in proposing a refined classification for the types of software evolution and software maintenance have been to:

- base the classification on objective evidence ascertainable from observation and/or a before and after comparison of the software, even when the knowledge of the personnel originally involved is no longer accessible;
- make the granularity of the proposed classification realistically reflect the actual mix of activities observed in the practice of software evolution and software maintenance;
- extend prior work on types of software maintenance [6] and on an ontology of software maintenance [3] to encompass software evolution as well as software maintenance;
- provide a realistic and practical classification to facilitate communication about, and the management of, software evolution and maintenance among researchers, practitioners, and their managers;
- supply a classification independent of the operating system, the hardware platform, the languages of implementation, organizational practices, design methodologies, and access to the personnel involved in doing the work; and
- offer enough consistency with existing terminology and definitions used in the field, to facilitate acceptance of our proposed classification and provide reasonable continuity.

Early published work classifying software maintenance captured primarily practitioner concerns. In 1972, Canning [7] summarized these in his landmark article "That Maintenance 'Iceberg"'. As he reported, practitioners saw maintenance narrowly as correcting errors, and broadly as expanding and extending software functionality. Some practitioners also included accommodating to changes in the underlying system software or hardware. To rationalize the practitioners' varied recognition of types of software maintenance, Swanson [8] in 1976 offered a typology of software maintenance. Swanson based his typology on the software system's owners'/users' dominant objective or intention in, or

'. . . basis. . . ' for, requesting or undertaking the maintenance work—i.e., what was seen as the cause or purpose of the maintenance, or why was the maintenance to be done [1,2,9]. As a mutually exclusive and exhaustive typology [10, p. 153], the three intentions for software maintenance Swanson pointed to were, in summary:

- to perfect the system in terms of its performance, processing efficiency, or maintainability ('perfective maintenance');
- to adapt the system to changes in its data environment or processing environment ('adaptive maintenance'); and
- to correct processing, performance, or implementation failures of the system ('corrective maintenance').

Such intentions, unless specifically recorded by the personnel involved at the time of the event, cannot in general be reliably and consistently determined after the fact from available objective evidence. Intentions reflect a mix of the character of the maintenance and the organizational and political context. Consider an example: In an organization, a user for over a year of a software-implemented system, for a second time initiates a change request asking for maintenance to correct a 'bug'—a functionality that is still not working as was specified in the user's requirements for the development of the system. The vice president with budgetary authority, who denied the prior change request for this same maintenance, approves the current change request as 'perfective maintenance', not because she believes it would correct anything, but because she believes there now is a real business need for the software to be improved to provide the requested functionality. Since the work involves modifying the system's GUI, the software maintenance manager assigns the change to the team deploying the new larger but more energy-efficient monitors, in order to give the team a realistic test case in its 'adaptive maintenance'. When the work is completed, what type of maintenance was it?

That said, an intentions basis for a maintenance typology is appropriate for and can be used in some kinds of valuable research, as has been well demonstrated, for instance, by Lientz and Swanson [10]. For example, Lientz and Swanson in 1980 found confirmation that software maintenance '. . . consists largely of *continued development*. . . ' [10, p. 151,11]. 'Continued development' that changes the system's functionality or properties as experienced by the customer of the system, is also commonly called 'software evolution' [12].

2. CURRENT STATE OF THE FIELD

Swanson's typology has been influential among researchers [13]. Many researchers have adopted the terminology ('corrective', 'adaptive', 'perfective') but few researchers have used the typology as Swanson defined it. Instead, they have given different meanings to the terms, and not agreed among themselves on the meanings. While this freshness has sometimes been invigorating, the lack of agreement on the meaning of commonly used terms has added to the chaos in the field by making it difficult for researchers to build upon each other's work. To its credit, the IEEE put out a glossary [14] that includes the terms 'corrective maintenance', 'adaptive maintenance', and 'perfective maintenance'. However, the definitions given in that IEEE *Glossary* are partially inconsistent with Swanson's definitions [8,14].

Table I. Approximate correspondence between definitions of types. Correspondence with the Chapin *et al.* definitions exists only when objective evidence confirms the activity. NI—not included; (All)—implicit in all included activities.

Evidence-based (Chapin *et al.*)		Intention-based definitions			Activity-based definitions	
Cluster	Type	Swanson [8]	IEEE [14,23]	ISO/IEC 14764 [27]	Kitchenham *et al.* [3]	ESF/EPSOM [33]
Support interface	Training	NI	NI	(All)	NI	User support
	Consultive	NI	NI	(All)	NI	User support
	Evaluative*	NI	NI	(All)	(All)	(All)
Documentation	Reformative	Perfective	Perfective	Perfective	(All)	Preventive
	Updative*	Perfective	Perfective	Perfective	(All)	Preventive
Software properties	Groomative	Perfective	Perfective	Perfective enhancement	Enhancements	Perfective
	Preventive	Perfective	Perfective, or Preventive	Preventive, or Perfective enhancement	Preventive‡	Anticipative or Preventive
	Performance	Perfective	Perfective	Perfective enhancement	Corrections, or Implementation change	Anticipative or Perfective
	Adaptive*	Adaptive	Adaptive	Perfective enhancement	Changed existing requirements‡	Anticipative or Adaptive
Business rules	Reductive	Perfective	Perfective	Perfective enhancement	Changed existing requirements‡	Evolutive
	Corrective	Corrective	Corrective	Corrective	Corrective	Corrective
	Enhancive*	Perfective	Perfective	Perfective enhancement	New requirements‡	Evolutive

*Default type within the cluster.
‡A subtype of Enhancements.

Nearly all researchers have made modifications, especially to the definitions, some explicitly such as [15–18] but most implicitly or tacitly such as [19–22], even when retaining the use of the terms 'corrective', 'adaptive', and 'perfective'. The IEEE 1998 standard on software maintenance [23] uses the three terms as part of the standard but for definitions it uses ones not fully consistent with either the Swanson [8] or the IEEE *Glossary* [14] terms. Some researchers and practitioners have drawn upon the ISO/IEC 12207 standard [24–26] that underlies the ISO/IEC 14764 standard [27], where the definitions are again different.

Other literature, as from industry, has followed a similar pattern, such as [28–30]. Terminology and definitions differ greatly between industries or domains, from organization to organization, and even within an organization (e.g., [31]). Organizations' in-practice typologies are frequently more detailed and recognize more types or activities than the three types in the Swanson typology. These more detailed typologies are useful in time and resource expenditure reporting, project progress reporting, cost allocating, budgeting, doing quality assurance, supervising programmers and analysts doing software maintenance, revising job descriptions, making skills inventories, hiring, planning staff needs, etc.

Such commonly encountered deviations in types and their definitions suggest that the time is ripe for a fresh classification of the types of software evolution and maintenance to meet the current concerns and needs of researchers and of practitioners and their managers. Serious efforts in that direction have either continued the intention-based approach, e.g. the ISO/IEC 14764 standard [27], or have drawn upon prior work on the activities and tasks involved in doing software maintenance, e.g. [1,2,6,32]. Among the major results has been the Euréka Software Factory's European Platform for Software Maintenance (ESF/EPSOM) [33], and the software ontology [3]. Table I summarizes half a dozen typologies. Even when the terminology, such as 'perfective', is the same in two or more typologies, the definitions of those terms are different in each with few exceptions.

In the next section (Section 3), we present our proposal, and follow that with its application in Section 4. After a discussion in Section 5 of matters asked about by reviewers of drafts of this paper, we close with conclusions in Section 6. Readers may find it helpful to understand our definitions of 25 terms as given in Appendix A.

3. PROPOSED CLASSIFICATION

3.1. Three criteria

We base our classification on work performed on the software of the evolved or maintained system, deliberately done as activities or processes, and observed, manifested or detected in:

- A—the software;
- B—the code; and
- C—the customer-experienced functionality.

Evidence of activities that produce these changes serves as the criteria; observation and comparison provide the evidence. The comparison is of the relevant parts of the software as of *before* the alleged evolution or maintenance, with the relevant parts of the software as of *after* the alleged evolution or maintenance. The observation is of the activities and their artifacts. None of the criteria require or

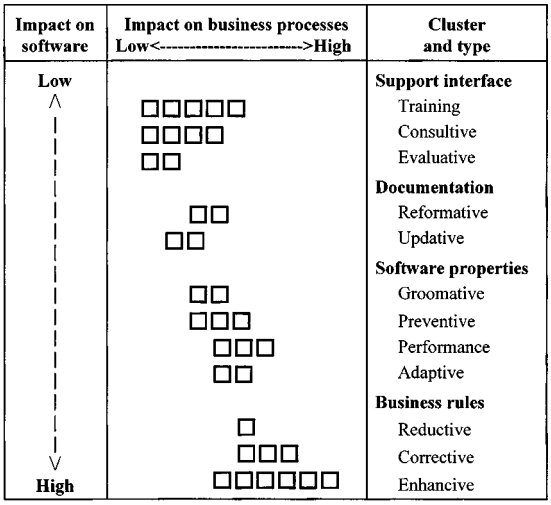


Figure 1. Impact of the types of software evolution and maintenance.

depend upon personnel's statements or recollections of intentions about, the causes of, the reasons for, or the purposes of the software evolution or software maintenance. But the reverse is sometimes true, that the objective evidence can sometimes be used to identify or infer the causes, purposes, intentions, reasons, or requirements for work done.

As the recent ontology has pointed out [3], software evolution or maintenance commonly involves several to many processes or activities that may result in from none to many changes in the software. From the set of all changes observed, made, or detected as attempted from the evidence, a dominant process or grouping of processes or activities typically emerges, with supporting or associated processes or groupings usually also present. We define our proposed classification to be exhaustive with mutually exclusive types, grouped into clusters, where any instance of software evolution or software maintenance may involve a mix or aggregate in any order of some or all of the types, even though one type may be deemed or observed as dominant.

The order of the types and their clusters is significant because of their different impacts, as Figure 1 illustrates. One dimension is the impact of the evolution or maintenance on the customer's ability to function effectively using the software-implemented system in meeting its goals—i.e., the business process impact on the customer—or how the customer does its/his/her business and the customer's satisfaction with the system [34]. This is drawn from left (low impact) to right (high impact). The number of blocks suggests the likely range of the business process impact on the customer. For instance, enhancing the software with new functionality for the customer typically has more effect on the customer's ability to meet its goals than does changing the non-code documentation. The other dimension is the impact of the evolution or maintenance on the software itself. This is diagrammed top

(low impact) to bottom (high impact). For instance, reformatting a page of a user manual typically has less effect on the software than does correcting (fixing) a bug.

3.2. Type clusters

In the form of a condensed decision tree, Figure 2 summarizes our proposal for an objective-evidence-based classification. The three criteria decisions—A, B, C—route to the appropriate cluster of types. Within each cluster, the type decisions characterize the type. Because of the significance patterns illustrated in Figure 1, the decision tree in Figure 2 is read from left to right for increasing impact on either or both the software and the business processes. The types are shown in four clusters, and within each cluster are read from bottom to top for increasing impact on either or both the software and the customer business processes.

Type decisions in the form of questions are asked about particular evidence. Note that in Figure 2, because of the limited space available, the wording of the decisions is abbreviated from the wording and explanations given here in the text of this paper. The associated types are shown in *italics* to the right of the type decisions and are only applicable when the associated responses to the respective type decisions are 'Yes'. Note that no decisions are asked about management matters or quality assurance. This is because we regarded relevant selections from those processes and activities as part of all of the types of software evolution and software maintenance.

Since software evolution or software maintenance typically involves many processes or activities, determining the type requires asking many of the questions in the decision tree. Progression to the right or higher in the decision tree leaves all to the left and lower as active candidate types. For example, if the maintenance done was the updating of the UML component diagrams but not the user manual, then the choice on the criterion decision A is 'Yes' and on B and C is 'No'. Hence, since no software functionality or properties were changed, then the types likely done in such a situation are *updative*, *reformative*, *evaluative*, *consultive*, or *training*, or some combination of them. One of these types may have dominated. If none dominated or a single type name was wanted, then the default choice is whatever 'Yes' answer to a type decision indicates the greatest impact (i.e., that which is farthest to the right and highest in that cluster). In this example, that cluster is 'documentation' and the type is *updative*. The highest impact type is also the default choice when the objective evidence to discriminate among the types is missing or moot for the type decisions within a cluster. When the objective evidence is missing or moot for the criteria decisions A, B, and C, the default choice is the 'No' alternative.

Note also that no type 'Other' is offered, because each cluster has a designated default type for use when the objective evidence is ambiguous. If some activity or process were done but what it was is ambiguous from observation and the documentation evidence, then the overall default type is *evaluative* because it typically is the most common type [1,35,36]. Of course, if observation and the other evidence indicate that no activities or processes were done to or with the software other than those involved in running it, then no type of software evolution or software maintenance occurred.

3.3. Support interface cluster

The A criterion decision 'Did the activities change the software?' is about the condition of the software before and after the activity. If we read the objective evidence as 'No' (e.g., the software was used only for reference), then we go to the **support interface cluster** consisting of type decisions A-1, A-2, and

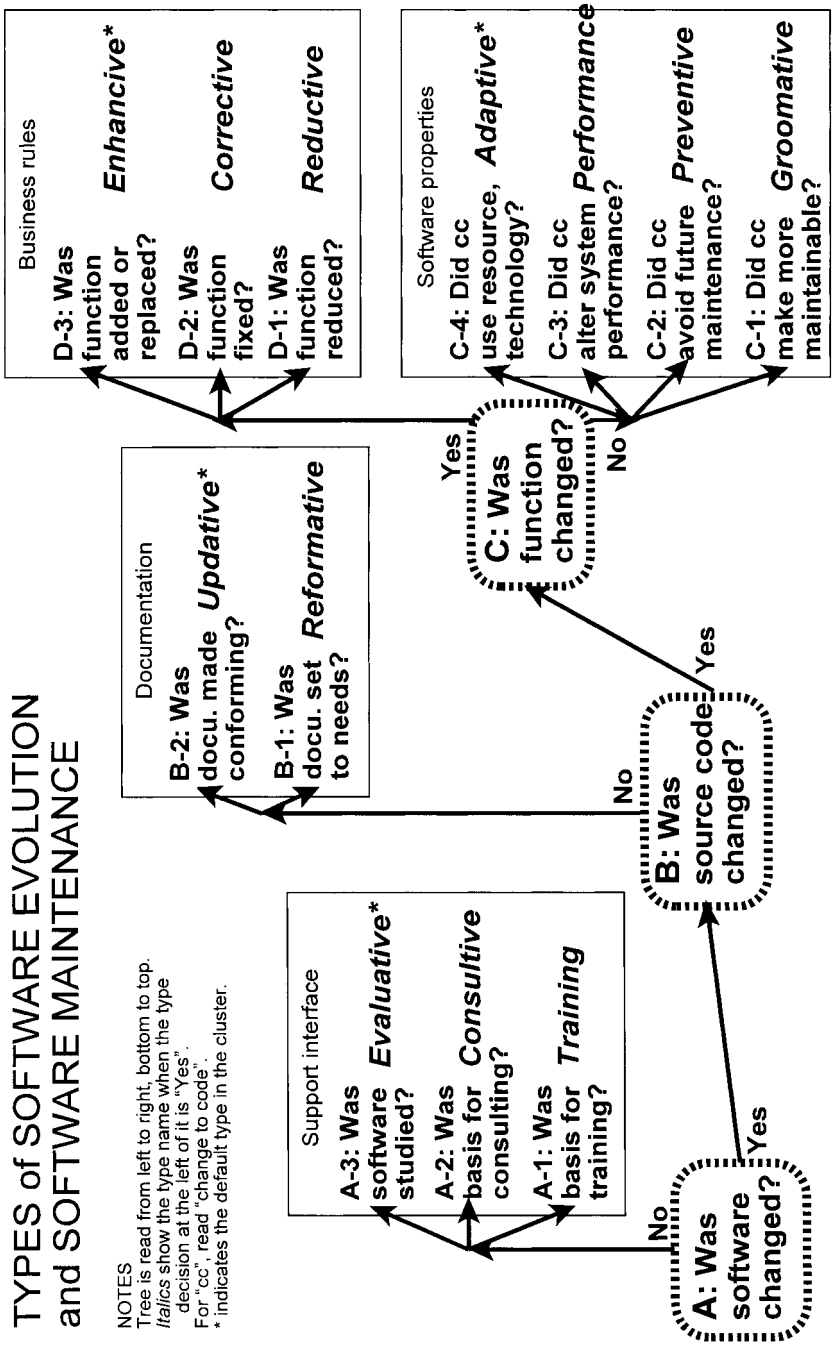


Figure 2. Decision tree for types.

A-3, which because of their increasing impact are considered in that order. *Evaluative* is the default type for this cluster.

The A-1 type decision is 'Did the activities use the software as a subject for stakeholder training?' This *training* type includes such activities as conducting classes for customer personnel, doing training ranging from on-site on-the-job to vestibule on-the-road training, and using existing training materials, since such materials are often derived from or are part of the documentation. Since changing organizational environments, and customer and other personnel turnover are inevitable, the importance of training has long been recognized [10].

The A-2 type decision is 'Did the activities use the software as a basis for consultation?' This *consultive* type is very common because it involves such activities as fielding questions at a help desk for customer personnel about the software, conferring with customer personnel or managers about the effective use of a system or about the making of (proposed or possible) changes to the software, and advising customers or managers or other stakeholders (or even researchers) about the likely time, cost, activities, or effort needed to do requested or contemplated evolution or maintenance work.

The A-3 type decision is 'Did the activities involve evaluating the software?' This *evaluative* type is a very common type because it involves such activities as auditing, searching, examining, regression testing, studying, diagnostic testing, providing protected testing environments, calculating metrics about, or creating an understanding of the software without changing the software ('getting on the outside of it'). Since in practice such evaluative processes typically account for more than half of the person-hours expended in software evolution and software maintenance [35,36], we propose that *evaluative* be deemed the appropriate dominant type only in instances where the aggregate of all of the other types of maintenance is trivially small. However, this type is the default type for the support interface cluster since it so commonly underlies other types.

3.4. Documentation cluster

If we read the objective evidence as 'Yes' for the lead-off A criterion decision (i.e., the software was changed), then we go to the B criterion decision about the condition of the code. It is important to note in making this B criterion decision that the code changed may not be in the system that was the original focus of the work but in an associated system or one that either supplies data to the system or receives data provided by the system. That is, the type may vary by system when a given effort ripples to or involves more than one program or system.

If we read the objective evidence as 'No' for the B criterion decision 'Did the activities change the code?', then we go to the **documentation cluster** consisting of type decisions B-1 and B-2, which because of their increasing impact are considered in that order. Activities in this cluster usually build upon activities in the software interface cluster.

The B-1 decision is 'Did the activities make the non-code documentation conform more closely to the stakeholder-stated needs?' This *reformative* type restyles or reformulates or refines the non-code documentation by changing its form while leaving the code untouched. This involves such activities as altering the format or coverage of a user manual, preparing training materials, incorporating the effects of a changed local standards manual, and changing the form or format of the non-code documentation by modifying the use of a CASE tool.

The B-2 decision is 'Did the activities make the non-code documentation conform more closely to the system as implemented?' This *updative* type involves updating the content and coverage, and

polishing the non-code documentation by such activities as replacing obsolete non-code documentation with current accurate documentation, and filling in gaps in the non-code documentation such as by incorporating test plans, narrative overviews, UML models, etc., without changing the code. This *updative* type is the default type for the documentation cluster.

3.5. Software properties cluster

If we read the objective evidence as 'Yes' for the B criterion decision (i.e., the code was changed), then we go to the C criterion decision that asks about the changes in the system functionality sensible by the customer. The C criterion decision is 'Did the activities change the customer-experienced functionality?' If we read the objective evidence as 'No' for the C criterion decision, then we go to the **software properties cluster** consisting of the C-1, C-2, C-3, and C-4 type decisions, which because of their increasing impact are considered in that order. In this cluster, note that the code changes do change the system or software properties—e.g., refactoring—but do not change functionality as experienced by the customer. Activities in this cluster usually build upon activities in the support interface cluster and trigger activities in the documentation cluster. *Adaptive* is the default type for the software properties cluster.

The C-1 type decision is 'Did the activities change maintainability or security?' This *groomative* type involves source code grooming activities, such as doing recompilations, replacing components or algorithms with more elegant ones, creating code backups, changing personal or level access authorizations, changing source code comments or annotation lines, changing data naming conventions, altering code readability or understandability, providing less-cryptic error messages, and preparing source code for pretty-printing. Such 'anti-regressive' [37] source code grooming activities yield current results that sometimes relate to the interaction of the system and its environment, and the supporting infrastructure and operating procedures, but do not alter performance properties. The results of such activities are rarely directly sensible by the customer. Their major impact is on maintainability. For instance, a major U.S.A. site that has routinely used inspections in its quality assurance processes found that 93% of the inspection-triggered changes to the source code improved maintainability, and that more than two-thirds of them were *groomative* [38].

The C-2 type decision is 'Did the activities avoid or reduce future maintenance activities?' This *preventive* type involves making changes that do not alter either the existing customer-experienced functionality or the existing technology or resources utilized. A recent example was much of the Y2K work done. The results of such preventive activities are rarely directly sensible by the customer. The preventive type sometimes is an administrative tool for managing work load or budget allocations or stakeholder relations. This type is sometimes done on a planned or 'scheduled' basis. Participation in software reuse efforts, some of which may involve using COTS, can be of the *preventive* type if they are relevant for an existing used system. Since the forecasting of future problems is usually imprecise and often only partially accurate, work that is classified when done as preventive may, with the benefit of subsequent additional evidence, be classified differently, usually as *groomative*, *adaptive*, or *enhancive*. The IEEE *Glossary* defines preventive maintenance in an intention-based manner from a hardware perspective [14, p. 57]. A six-person panel at ICSM 2000 focused on software preventive maintenance, but did not reach a consensus on what it is or how to define or measure it [39].

The C-3 type decision is 'Did the activities alter software performance characteristics or properties?' This *performance* type involves such processes as reducing the amount of internal storage used,

improving system up time, reducing the duration of out-of-service periods, speeding execution as by replacing components or algorithms with faster ones, and improving system reliability and robustness. The change in such properties is often sensible by the customer. This type is sometimes done on a planned or 'scheduled' basis.

The C-4 type decision is 'Did the activities change the technology or resources used?' This *adaptive* type involves such processes as making the system work on a different platform or with a changed operating system or with a different kind of database, reallocating functions among components or subsystems, increasing COTS utilization, changing communication protocols supported, altering the system's interoperability, and changing design and implementation practices such as moving to object-oriented technology. Such activities often change system properties sensible by the customer. Note that this adaptive type does not change the existing customer-experienced functionality, but, like the other types in this cluster, does change the system properties or characteristics. This type is sometimes done on a planned or 'scheduled' basis. In this software properties cluster, *adaptive* is the default type.

3.6. Business rules cluster

If we read the evidence as 'Yes' for the C criterion decision (i.e., the customer-experienced functionality was changed), then we go to the **business rules cluster** consisting of the D-1, D-2, and D-3 type decisions, which because of their increasing impact are considered in that order. The activities here are among the most frequent and most significant in software evolution and maintenance, and usually rely upon an extensive supportive use of activities from the other clusters. All three types are sometimes done on a planned or 'scheduled' basis. The default type in this cluster is *enhancive*.

The D-1 type decision is 'Did the activities restrict or reduce the customer-experienced functionality?' This *reductive* type, of particular significance for software evolution, involves limiting or eliminating some business rules from the implemented repertoire, as by replacing or constraining or removing part or all of some components or algorithms or subsystems. Data flows may be eliminated or reduced. This type as a dominant type is relatively rare, especially with embedded software because of the installed base, and when dominant is most often done in support of an organizational spin-off or merger when as a consequence a reduced set of business rules are thereafter used by the customer.

The D-2 type decision is 'Did the activities fix the customer-experienced functionality or make it more correct?' This *corrective* type involves refining and making more specific the implementation of the existing business rules to handle exceptions and mishandled cases better, usually by adding additional precision to the internal logic of some components or algorithms or subsystems, or by adding more defensive programming. This reactive 'fix-it' type removes defects ('bugs') such as those from design shortfalls or coding errors. The type is restorative of the prior customer-experienced functionality by compensating for invalid assumptions and for oversight ripple changes in the data handled by the system's interacting systems and supporting infrastructure, and sometimes by compensating for changes in the customer's milieu that have changed data availability. The corrective type is done sometimes on a request basis, sometimes ad hoc on an 'emergency' basis, and sometimes (as noted above) on a planned or 'scheduled' basis. A detailed taxonomy covering some of the activities done in the *corrective* type is given in [40].

The D-3 type decision is 'Did the activities replace, add to, or extend the customer-experienced functionality?' This *enhancive* type, of particular significance for software evolution, implements changes and additions to the repertoire of software-implemented business rules, as by inserting new

components, algorithms, or subsystems, and altering existing ones to enlarge or extend their scope. New data flows may be added and existing data flows may be redirected. The changes may affect any part of a system's functionality as experienced by the system's customers, including the levels of data digestion or abstraction, and privacy and security facilities. Because of the typically increasing complexity of customers' operations, the *enhancive* type is usually the most common and most significant, subject to the caveat noted earlier about the *evaluative* type. *Enhancive* is the default type for the business rules cluster.

4. APPLICATION

Table I, presented previously, summarizes a comparison of our proposal with five other ways of distinguishing types of software maintenance. Our proposal is a classification based upon objective evidence about activities.

Maintainers do their work differently in different organizations and in different work assignments. Hence, in seeking activities to observe and in seeking objective evidence of activities for identifying types of software evolution and software maintenance, some preparation can save time and improve accuracy. Three preparation actions are normally worthwhile:

- ascertaining what activities are specified by the local standard operating practices (SOP) manual or equivalent if any, and the extent to which its dictates are and have been enforced;
- determining what are and have been the documentation practices; the documentation situation is often similar or tied to the activities noted in the SOP;
- obtaining access to relevant documentation; documentation may be centralized or dispersed, hard copy or soft, existent or missing, filed, loaned out, in use, archived, etc.

With a variety of activities carried out, many of the types of software evolution and software maintenance can to some degree be observed, even when some particular type appears to be dominant. The question then is what are the relationships among the types? Figure 3 illustrates the relationships in a simple manner, where those higher up are more significant because they have more impact on the customer or the software or both. The key to understanding the relationships goes back to the three criteria decisions A, B, and C. All three must always be asked. Responding 'No' to any one of them identifies a respective cluster of types where at least one of the type decisions may earn a 'Yes' for some type. It also for the A and B criteria decisions takes us on to ask the next criterion decision, B or C respectively. In addition, responding either 'No' or 'Yes' to either the B or C criteria decision is also effectively invoking the 'No' response cluster associated with the A or B criteria decisions respectively.

While that explanation sounds complicated, the process is simple—it is just how far we ride up in Figure 3, which is equivalent to sweeping through portions of the tree in Figure 2. Consider an example. A maintainer, after studying the documentation including the source code, rewrote the source code for one module, and without changing any other documentation, compiled the revised module, tested it, and installed it in the production library. The only consequence the customer saw was a faster response time.

Question: what type of software evolution or software maintenance was this? From the evidence reported, criterion A is 'Yes', criterion B is 'Yes', and criterion C is 'No'. In the properties cluster, the evidence gives us a 'Yes' for the *performance* type in the software properties cluster. But the evidence also points to the *evaluative* type and it is in the support interface cluster. Note also that this example

Type		Class	
Cluster	**Specific type**	**E**	**M**
Business rules	Enhancive	E	M
	Corrective		
	Reductive		
Software properties	Adaptive	E	M
	Performance		
	Preventive		M
	Groomative		
Documentation	Updative		M
	Reformative		
Support interface	Evaluative		M
	Consultive		
	Training		

Figure 3. Relationships among types, where E indicates software evolution, and M indicates software maintenance.

does also qualify as software evolution as well as software maintenance. In Figure 3, we locate the two types, and note that *performance* is higher than *evaluative*. This means that we would have expected in this example to see evidence of the *training*, *consultive*, *reformative*, *updative*, *groomative*, and *preventive* types, but no evidence of the *adaptive*, *reductive*, *corrective*, or *enhancive* types.

This helps us ask potentially fruitful questions. Are the missing types missing because of oversight in observation or in examining the objective evidence, or because they indicate activities not done? Was it right that no customer training was needed, or that neither the supervisor or the customer was consulted with? How was the work authorized? Has the supervision of the maintainer been lax? Have the directions in the local SOP manual been slighted? Why have the test results not been put into the documentation? Was the budget or schedule too lean to get all the work done? Was there a follow-on project to fill in the missing pieces, like updating the documentation? With work like this, what is the effect upon software maintainability, and is that what the organization wants?

Consider another example. Two maintainers were assigned to take care of a change request. They examined the system to estimate what portions would be affected for the customer's proposed changes, and then conferred with their manager and the customer about the likely level of effort (cost and time) required. The result was some negotiation with their manager and the customer and about the deadline and scope of the project before the approval came to go ahead. The maintainers first put in time in building comprehension of both the client and the two servers comprising the system, updated part

of the documentation, replaced a system chart with a UML static view, and generated some new data layouts. With the to-be-changed portion of the client software targeted, the maintainers cleaned up an instance of initialization coding without changing its function, obtained faster response from the database server by changing the date representation needed for access, and reduced the performance delays the customer had expressed some concern about by increasing the application server's queue's capacity. Focusing on the customer's proposed changes, the maintainers deleted the code that treated returning former suppliers separately, and inserted data definitions and code to process data on a new class of suppliers. Also, the maintainers fixed the irregular truncation of place names the customer had complained about in passing. Finally, the maintainers prepared a change to the user manual, and showed the customer liaison person how to be sure that instances of the new class of suppliers were properly flagged. After compiling, testing, and debugging the software, the maintainers ran regression tests in order to get acceptance by SQA (software quality assurance) and IS (information systems) auditing. The maintainers informed their manager that the assignment was ready for release, and that with their manager's approval, the modified software was ready to be put into production with an update of the configuration management data. The maintainers also reported to their manager a less than a 1% increase in system size and no change in the average cyclomatic number for the software modules.

Question: what type of software evolution or software maintenance was this? All three responses on the criteria decisions are 'Yes' so the example qualifies as both software evolution and software maintenance. When we examined the documentation before and after, what stood out most? From the sketch given and using Figures 2 and 3, we can find evidence of 11 types with only the *preventive* type not supported by the evidence. Even though getting a grasp on the code and regression testing the work appeared to have accounted for the most person-hours (*evaluative*), what had the most software and business impact was the minor increase in functionality gained by the customer. Therefore, within the business rules cluster, we seek the type that yields the best fit with the evidence. In this example, *enhancive* is the dominant type of both software evolution and software maintenance. Note, however, that both practitioners and their managers, as well as researchers, may be interested in all of the types individually identifiable in any given work situation, for the insight and management opportunities they may reveal, such as noted in the previous example.

Finally, consider still another example. A maintainer was assigned to install three upgraded COTS components, the first of which implements a hardware upgrade. Each COTS component was received from a different vendor, but all are used in one system. In attempting the installation on the test version of the system, the maintainer found that one of the upgrades was incompatible with the other two, and that those other two would not work with the existing in-use version of the third. After considerable diagnostic testing runs, and obtaining an 'its your problem' response from the vendor of the third component, the maintainer got approval to write a wrapper for one of the upgraded COTS components in order to fit with the continued use of the existing version of the third component. After successful regression testing, the maintainer had the configuration management data updated and the new wrapper recorded. The change, since it would be nearly transparent to the customer, was put into production at the time of the next scheduled release of a version of the system, and the documentation was updated.

Question: what type of software evolution or software maintenance was this? The responses on the three criteria questions were A 'Yes', B 'Yes', and C 'No'. The evidence supports the *consultive*, *evaluative*, *updative*, and *adaptive* types of software maintenance as having being done, and qualifies as an instance of software evolution. As Figure 3 indicates, the dominant type here was *adaptive*. Management is likely to be interested in seeing how the personnel time was spent across the various

types in maintaining this COTS-using system. A finer granularity view can help in the management of the personnel skill resources available in relation to other resource availability and the demands from other work being done.

We offer Appendix B as an assistance in assessing observations on software evolution and software maintenance activities and the documentation evidence for such activities. Appendix B is organized in three sets of columns. The left column shows the type name, the middle column shows ticks for each relevant cluster, and the right column lists the activity. Appendix B is ordered alphabetically by the activity, and the listing is far from exhaustive. Since activities can be described in multiple ways, sometimes the same activity is listed in different ways, such as 'regression testing' and 'testing, regression', or such as 'comprehending the system' and 'grasping the system'.

5. DISCUSSION

The term 'evolution' has been used since the early 1960s to characterize the growth dynamics of software. Halpern [11] in 1964 applied it to software implementing programming systems, for example. The term 'evolution' in relation to application systems took hold more slowly, with an early mention being in 1970 and a paper being published the next year [41]. In 1985, Lehman and Belady, keying off their study of several major systems, identified five laws of evolution as intrinsic to 'E-system' software—i.e., software being used in real-world domains and having stakeholders—because of evolutionary pressures present during development and during the continuing processes aimed at maintaining the system and its cost-effectiveness and acceptability to its stakeholders within a changing environment [34]. Responding to these pressures typically results in an asymmetric or clumpy growth in system functionality [42]. Software evolution has been further studied and three additional laws describing its phenomenology at a high level have been advanced [43]. Low-level and mid-level descriptions of it have drawn primarily from studies of software maintenance [44,45]. Some studies have focused on techniques for accomplishing software change (the 'how'), and others on the phenomena of software change and its drivers and consequences (the 'why') [46].

As Bennett and Rajlich have pointed out, software evolution currently has no one widely accepted definition, but some researchers and a small proportion of practitioners currently use the term 'software evolution' as a preferred substitute for the term 'software maintenance' [19,47]. Others prefer a narrow view that software evolution occurs when either *enhancive* or *reductive* maintenance is carried out [34,43]. The current common view ([12], and see Figure 3) is that software evolution occurs when the software maintenance done is of the *enhancive*, *corrective*, or *reductive* types (any of the business rules cluster), or it changes software properties sensible by the customer, i.e., it is of the *adaptive* or *performance* types. This objective-evidence basis for identifying software evolution is, we believe, easier to apply and generally more useful than an intentions-based typology.

The term 'maintenance' has been a term used for making deliberate modifications of the software part of systems implemented at least in part with software, since the early 1960s. The terms 'change' or 'modification' were common if the activities were done by the personnel who had done the original development of the software [48,49]. 'Maintenance' was associated with change or modification made by other personnel to existing software for which they had not carried out the development. In the 1973 to 1975 time frame, IBM reorganized the way it supported making changes to its software products and helping customers use them, largely freeing the development personnel from subsequent support activities. That model has continued to influence the group of activities that comprise 'software

maintenance', although many other terms have been and are being used as substitutes for 'software maintenance', such as 'operational support' or 'continuous development'.

Both increased competition and the Internet are increasing the need for fast software evolution and software maintenance. As time to market and organizational responsiveness become more important, software modification activities and processes using them are being adapted to gain faster change. However, just substituting new current data into the software supporting the web site (effectively just changing values of the input data for processing) no more qualifies as either maintenance or evolution than does handling batch data arising from sales paid for by credit card tenders.

Sometimes quick maintenance is required. For example, an organization active in international politics may have to modify how it presents data on its web site several times during a day. To the webmaster and the maintainer support staff, the situation often appears like continuous change in the software, with releases made on-the-fly. What the customer can access from one hour to the next may be very different, including its form and content, with still or video graphics, sound or none, animation or none, changed colours, and in varied layouts. Less severe but usually at a larger scale are business or government organizations that have to make fast responses to changing environmental or competitive conditions, such as in modifying systems to support new products or services, and revised terms of product or service availability. Without updated software meeting customers' current needs, the organization becomes vulnerable to falling behind its competition [50]. Our proposed types of software evolution and software maintenance are as applicable to such quick response software evolution and software maintenance situations as they are to the common traditional situations and to embedded software modification.

Our proposed types of software evolution and software maintenance can extend, as noted earlier, into what has been called software 'development'. For example, while producing the initial version of a system, correcting an oversight logic bug detected during unit testing qualifies as *corrective* software maintenance (the correction was to existing software), but not as software evolution (the correction was in a first version of the system and not used by the customer). Narrowly, development has been the initial creation of a system implemented with software written from scratch. Yet in practice in industry, business, and government a broader usage is common. The broader usage includes creating systems assembled from components (such as COTS) with some interactions among the components mediated by freshly written middleware, wrappers, or 'glue code'. And the broader usage also includes classifying as development all software projects estimated to require more than some specified amount of either person time or calendar time. We have seen the person-time threshold set as low as half a person-day, and the calendar-time threshold set as low as one calendar day. Low thresholds result in officially classifying most software maintenance projects as instances of development, but in terms of the activities done and what they are done on or with, they are still actually software maintenance.

Business rules pervade software [51,52]. While this is easily and widely recognized in some specific domains, such as in payroll processing, it is far less obvious in many others. For example, the management and performance of a GUI are mostly implementations of business rules. When expressed in software, a business rule has a simple underlying basic structure: a logical case predicate leading to particular transformations. Each part becomes elaborated and customized to the situation, with nesting common. In general, the data for the predicate have to be accessed and their values validated. Each transformation takes in validated data ultimately traceable to the system's input, and produces data ultimately traceable to the system's output. Hence, changes in the software expression of business rules results in changes in the customer-experienced functionality, and they are its dominant effector.

Because business rules affect how a customer accomplishes its/his/her mission, task, or function, they may have proprietary or even trade secret status. Our proposal classifies the activities making such changes into three types, *reductive*, *corrective*, and *enhancive*, all in the business rules cluster.

Our proposed types of software evolution and software maintenance are independent of one of the concerns expressed in the ontology report [3], the matter termed there as 'scenarios'. The types proposed here cut through not only the two scenarios treated in that report, but also nearly all of the possible scenarios, assuming that neither total outsourcing nor the combination of total retirement and replacement are regarded as either evolution or maintenance.

Also, what the ontology report [3] terms 'modification activity' is found in the types comprising our documentation, software properties, and business rules clusters. What the ontology paper terms 'investigation activity' is split in our proposal. The general investigation activities are in the *evaluation* type in the support interface cluster. The focused ones, depending upon their focus, are in the types comprising our documentation, software properties, and business rules clusters.

The choice of names for the types of software evolution and software maintenance has been a matter of concern. The 'support interface' cluster, for example, got its name because some of the activities in the types are at the interface between the maintainers and other maintainers, managers of maintainers, and the system stakeholders, especially the customers. Projects are started and finished in these activities, and the continuing supportive training, assistance, and consultation are here too.

On names for the types, we followed the lead of Swanson [8], preferring to use adjectives ending in . . . ive. This has led us to coin a term 'groomative', that we offer as an adequately descriptive adjective for grooming the software. Also, we have dropped 'perfective' to make our proposal more precise and useful both to practitioners and their managers, and to researchers. We recognize that previous attempts at the redefinition of terms have been widely ignored, as for example [14,15,28]. We hope that our proposal will be better accepted and be found to be useful.

Also, we have redefined the meaning of two terms used by Swanson [8], 'adaptive' and 'corrective' by eliminating the intention component and narrowing the applicability to software properties and business rules respectively. We propose these two redefinitions in spite of realizing that the many previous attempts to modify the intentional definitions of 'adaptive' and 'corrective' have not been greeted with success, and that our proposal may contribute to further confusion. We persist in these two redefinition proposals for three reasons:

- we believe the acceptance of two newly coined terms for these types of maintenance would probably gain even less acceptance;
- the terms help provide some continuity with popular usage, as noted earlier; and
- these two types, *corrective* and *adaptive*, along with *enhancive*, get the most attention.

Furthermore, we have not used the term 'features' because its current varied meanings seem well entrenched. Thus, a feature may be an aspect of the system functionality experienced by the customer. It may be a group of software behaviours. It may be a system property, such as running under a particular version of a particular operating system. A feature may even be an unfixed software defect or 'bug' in the view either of some customers or of some software suppliers.

Finally, we point out a major limitation of our proposal. As a detailed analysis can easily show, the use of the proposed criteria decisions even in their three levels can lead to far more than just a dozen types, if additional type decisions be inserted. With the classification we propose, specialized situations will likely warrant identifying subtypes. For example, one manager told us he considers *evaluative* as a

combination of three subtypes: familiarization, testing, and annual system auditing, because these are relevant in his organization in the management and accounting or personnel time utilization. Identifying subtypes permits customization that can make our proposal more flexible and useful.

6. CONCLUSIONS

In this paper on types of software evolution and software maintenance, we have proposed:

- recognizing types based on objective evidence, instead of based on intentions;
- pointing to the software, the code, and the customer-experienced functionality as key determinants of the types;
- providing a somewhat finer granularity for the types, with four named general types (as 'clusters') instead of the current total of three types (corrective, adaptive, and perfective);
- providing a detailed granularity for the types, with 12 named types;
- identifying observable activities or evidence that distinguish the 12 named types;
- extending type applicability to both software evolution and software maintenance;
- organizing the types to reflect their business process and software impacts; and
- approaching the classification of activities in an illuminating way for practitioners and their managers, as well as for researchers.

We rely on a common observation for the defence of our choice of objective evidence about activities as the basis for the classification. Carrying out different activities well calls upon different skill sets. In practice, because of human variability, different people with their different skill sets achieve different results, different qualities of results, and they achieve them with different productivities and at different speeds. Results, quality, productivity, and time taken are key concerns of maintainers, their managers, and empirical researchers. Our types proposal helps these people communicate more effectively about those key concerns, as we have shown in Section 4.

While we summarize in Appendix A our working definitions of some terms, our proposal in this paper does more than distinguish types of software evolution and software maintenance. It also effectively offers definitions of both software evolution and software maintenance as being processes which are recognizable, given the context, from their constituent activities.

APPENDIX A. DEFINITIONS OF TERMS

We found that some definitions helped us communicate more clearly in working out and using our proposed classification. We summarize our working definitions below:

- *software* or *computer software*—the non-hardware part, including associated documentation, of a system being implemented or implemented in part with a computer or an embedded processor [53];
- *software maintenance* or *maintenance*—the deliberate application of activities and processes, whether or not completed, to existing software that modify either the way the software directs hardware of the system, or the way the system (of which the software is a part) contributes to the business of the system's stakeholders, together with the associated quality assurance activities and processes, and with the management of the activities and processes, and often done in the context of software evolution (note that, in contrast to the IEEE *Glossary* definition [14] and the

IEEE maintenance standard definition [23], 'post-deployment' or 'after-delivery' status for the software is not part of our definition; activities involving existing software during 'development' before its delivery (sometimes called turnover or transition) qualify as software maintenance activities [54], but may not qualify as software evolution activities);

- *software evolution*—the application of software maintenance activities and processes that generate a new operational software version with a changed customer-experienced functionality or properties from a prior operational version, where the time period between versions may last from less than a minute to decades, together with the associated quality assurance activities and processes, and with the management of the activities and processes; sometimes used narrowly as a synonym for software maintenance, and sometimes used broadly as the sequence of states and the transitions between them of software from its initial creation to its eventual retirement or abandonment;
- *documentation*—the human-readable text and/or graphics (tangible such as on paper and intangible such as displayed on a monitor) specifying or describing computer software (e.g., requirements, data layouts, configuration management listings, regression test plans, change requests or work orders, user manuals, source code, etc.);
- *non-code documentation*—all documentation except the code;
- *source code*—this subset of the documentation is the human-readable form of the directions to the computer or processor that can be translated, compiled or interpreted to produce object code, microcode, or machine language useable by the hardware in producing the functionality and performance of the system;
- *code*—the source code and/or object code and/or machine code and/or microcode;
- *stakeholder*—the person, persons, part or parts of an organization, organization, or customer that has an interest in and may be affected by the functionality or performance of a system;
- *customer*—the person, persons, part or parts of an organization, or organization that uses the functionality or performance of a system for satisfying its goals, mission, or function (i.e., accomplishing its business), and is usually the primary stakeholder of the system;
- *user*—a customer or one of the personnel of a customer;
- *customer-experienced functionality*—the collection of system functions apparent to the customer; or the features and facilities provided to the customer by the system and implemented through the data provided by or on behalf of the customer to the system, and the data provided to or on behalf of the customer from the system, together with the timing and conditions under which the data flows occur between the system and the customer (such as responding to a query about the quantity of obsolete stock on hand); as distinct from data and/or services provided to or done by or for the system undetectable by the customer (such as the lengthening of an internal stack from a push of a subroutine address onto it);
- *business*—the work a person or organization does, such as a profession or occupation, and with which the person or organization is occupied, such as doing research, collecting taxes, measuring solar flares, refining petroleum, etc. (note that we do not limit the use of the term 'business' to only for-profit work such as selling souvenirs to tourists);
- *business rule*—a step or set of steps in a process or procedure or guide (algorithmic or heuristic) used by a customer for doing its business, work, or function, and often embodied in whole or in part in the software of a system [51,55], as for example, how much of a particular stock item to reorder and when to reorder it, or how to select the font style for names in a research report; and

- *software property*—a static or dynamic characteristic or attribute of a system or of its performance, such as its speed, its size, its language of implementation, its design methodology, its use of information technology resources, its recovery facility, etc.; sometimes called system property or system characteristic.

APPENDIX B. TYPES FOR SOME FREQUENTLY OBSERVED ACTIVITIES

In Table B1, the activities are listed in alphabetic sequence in the right-hand column. The columns to the left show the cluster and the type within that cluster for each item in the right-hand column. The list of activities is far from exhaustive; only some of the more frequently observed activities are listed. The word 'functionality' is used as a shortening of the phrase 'customer-experienced functionality'. The clusters are: SI—support interface, D—documentation, SP—software properties, and BR—business rules.

Table B1. Type and cluster for some frequently observed activities.

	Cluster				
Type	SI	D	SP	BR	Activity
					A
Enhancive	—	—	—	✓	Adding component or algorithm implementing functionality
Enhancive	—	—	—	✓	Algorithm replacement, adding or changing functionality
Groomative	—	—	✓	—	Algorithm replacement, functionality, properties unchanged
Performance	—	—	✓	—	Algorithm replacement, properties added or changed
Needs specifics	—	—	—	—	Analysing done (as a part of what activities?)
Groomative	—	—	✓	—	Annotation in code, revision, or addition
Consultive	✓	—	—	—	Answering questions about the software
Consultive	✓	—	—	—	Assisting personnel about the software
Corrective	—	—	—	✓	Assumption modified or replaced
Evaluative	✓	—	—	—	Auditing the software or information system
Groomative	—	—	✓	—	Authorizing changes for personnel security
Preventive	—	—	✓	—	Avoiding future maintenance by taking pre-emptive action
					B
Updative	—	✓	—	—	Backup creation or replacement or update of documentation
Corrective	—	—	—	✓	Bug fixing where code deemed to have been working OK
Enhancive	—	—	—	✓	Business rule added
Corrective	—	—	—	✓	Business rule fixed that had been deemed as working OK
Reductive	—	—	—	✓	Business rule removed
Enhancive	—	—	—	✓	Business rule replaced
					C
Evaluative	✓	—	—	—	Calculating metrics as on complexity, through-put, etc.
Reformative	—	✓	—	—	CASE tool usage change for non-code documentation
Training	✓	—	—	—	Class instruction, done or received, from existing materials
Groomative	—	—	✓	—	Code clean up or styling change
Updative	—	✓	—	—	Code backup creation or replacement or update

Table B1. Continued.

Type	Cluster SI	D	SP	BR	Activity
Needs specifics	—	—	—	—	Coding done (as a part of what activities?)
Corrective	—	—	—	✓	Coding error fixed where had been deemed as working OK
Groomative	—	—	✓	—	Comments in the code, added or revised or restyled
Adaptive	—	—	✓	—	Communication protocols changed
Enhancive	—	—	—	✓	Components replacement, adding or changing functionality
Groomative	—	—	✓	—	Components replacement, functionality, properties unchanged
Performance	—	—	✓	—	Components replacement, properties added or changed
Evaluative	✓	—	—	—	Comprehending the software, system, work order, etc.
Consultive	✓	—	—	—	Conferring about software, work order, or change request
Groomative	—	—	✓	—	Configuration file changed
Evaluative	✓	—	—	—	Configuration management data changed, corrected, updated
Updative	—	✓	—	—	Content conforming change made in non-code documentation
Needs specifics	—	—	—	—	Conversion of data format, layout, data codes (part of what?)
Evaluative	✓	—	—	—	Conversion of system status from development to maintenance
Enhancive	—	—	—	✓	COTS component functionality changed
Groomative	—	—	✓	—	COTS component, no change in functionality or technology
Performance	—	—	✓	—	COTS component performance changed
Adaptive	—	—	✓	—	COTS component, technology or resources changed
Reformative	—	✓	—	—	Coverage of user manual, non-code documentation changed
					D
Adaptive	—	—	✓	—	Databases changed or added
Enhancive	—	—	—	✓	Data flow added or redirected
Reductive	—	—	—	✓	Data flow eliminated or reduced
Corrective	—	—	—	✓	Data flow fixed that had been deemed as working OK
Needs specifics	—	—	—	—	Data layout done (as a part of what activities?)
Needs specifics	—	—	—	—	Debugging (as a part of what activities?)
Needs specifics	—	—	—	—	Defensive programming or defensive design (use on what?)
Corrective	—	—	—	✓	Design defect fixed where had been deemed OK
Adaptive	—	—	✓	—	Design methodology changed, code changed to implement
Needs specifics	—	—	—	—	Designing done (as a part of what activities?)
Evaluative	✓	—	—	—	Diagnostic testing
Needs specifics	—	—	—	—	Documenting done (as a part of what activities?)
Updative	—	✓	—	—	Documenting done with kind undetermined
Performance	—	—	✓	—	Down time frequency or duration or severity changed
Performance	—	—	✓	—	Duration of out-of-service changed
					E
Evaluative	✓	—	—	—	Environment for testing set up or changed
Corrective	—	—	—	✓	Error fixed where code had been deemed as working OK
Groomative	—	—	✓	—	Error message changed
Consultive	✓	—	—	—	Estimating working time, cost, resources, or computer time
Needs specifics	—	—	—	—	Examining test results (what kind of testing?)
Evaluative	✓	—	—	—	Examining the software, work order, operating situation, etc.
Enhancive	—	—	—	✓	Exception handling changed

Table B1. Continued.

Type	Cluster SI	D	SP	BR	Activity
Corrective	—	—	—	✓	Exception handling fixed where had been deemed OK
Adaptive	—	—	✓	—	Execution mode changed (e.g., to distributed)
Performance	—	—	✓	—	Execution speed changed
Performance	—	—	✓	—	External storage utilization changed
					F
Evaluative	✓	—	—	—	Familiarization with the software, work order, resources, etc.
Evaluative	✓	—	—	—	Field testing done
Reformative	—	✓	—	—	Form of non-code documentation changed
Preventive	—	—	✓	—	Future maintenance avoidance by pre-emptive action
					G
Updative	—	✓	—	—	Gaps, filling in non-code documentation
Evaluative	✓	—	—	—	Gathering data about the system and its software
Evaluative	✓	—	—	—	Getting on the outside of the software or work order
Updative	—	✓	—	—	Graphic non-code documentation updating
Evaluative	✓	—	—	—	Grasping the system
Groomative	—	—	✓	—	Grooming the code
Reformative	—	✓	—	—	Grooming the non-code documentation
					H
Consultive	✓	—	—	—	Help desk service provided
Evaluative	✓	—	—	—	Handoff or handover of system to maintenance
					I
Enhancive	—	—	—	✓	Inserting component or algorithm implementing functionality
Evaluative	✓	—	—	—	Install for operation or test run
Training	✓	—	—	—	Instruction done or received from existing materials
Needs specifics	—	—	—	—	Integration testing (properties or business rules?)
Performance	—	—	✓	—	Internal storage utilization changed
Adaptive	—	—	✓	—	Interoperability changed
Corrective	—	—	—	✓	Invalid assumption modified or replaced
					J
Consultive	✓	—	—	—	Joint maintenance activities with customer personnel
					K
Needs specifics	—	—	—	—	Kick out of errors (what kind of testing?)
					L
Groomative	—	—	✓	—	Level of security authorization change
Groomative	—	—	✓	—	Linkage library file changed
Reformative	—	✓	—	—	Local SOP manual change, adapting non-code documentation
Groomative	—	—	✓	—	Local SOP manual change, code styling

Table B1. Continued.

Type	Cluster SI	D	SP	BR	Activity
					M
Consultive	✓	—	—	—	Making estimates for managers, customers, etc.
Groomative	—	—	✓	—	Message changed about errors
Evaluative	✓	—	—	—	Metrics collection or calculation
Updative	—	✓	—	—	Model revision in non-code documentation
Adaptive	—	—	✓	—	Mode of execution changed (e.g., to distributed)
Performance	—	—	✓	—	Mode of use changed (e.g., to multi-user)
					N
Updative	—	✓	—	—	Narrative documentation updating
Updative	—	✓	—	—	Non-code documentation, making conform to code
Reformative	—	✓	—	—	Non-code documentation, making fit customer stated need
					O
Needs specifics	—	—	—	—	Obsolete code removal or replacement (what was the effect?)
Updative	—	✓	—	—	Obsolete non-code documentation removal or replacement
Training	✓	—	—	—	One-on-one teaching
Consultive	✓	—	—	—	On-the-job assistance given or received
Consultive	✓	—	—	—	On-site assistance provided or received
Adaptive	—	—	✓	—	Operating system changed
Performance	—	—	✓	—	Out of service durations changed
Consultive	✓	—	—	—	Outsourcing of maintenance, advising on
(mgt. assumed)	—	—	—	—	Outsourcing of maintenance, management of
Consultive	✓	—	—	—	Outsourcing of maintenance, routine interaction with
					P
Adaptive	—	—	✓	—	Platforms changed, added or discontinued
Groomative	—	—	✓	—	Polishing the code
Reformative	—	✓	—	—	Polishing the non-code documentation
Reformative	—	✓	—	—	Preparing or revising training materials re a system
Groomative	—	—	✓	—	Pretty-printing of code
Preventive	—	—	✓	—	Problem avoidance by taking pre-emptive action
Training	✓	—	—	—	Product training, done or received
Evaluative	✓	—	—	—	Program comprehension
Corrective	—	—	—	✓	Programming error fixed where been deemed as working OK
Adaptive	—	—	✓	—	Protocols changed
					Q
Consultive	✓	—	—	—	Query response, determining or making
					R
Groomative	—	—	✓	—	Readability, changing the code for
Reformative	—	✓	—	—	Readability, changing the non-code documentation for
Reformative	—	✓	—	—	Recompilation to get fresh source listing
Adaptive	—	—	✓	—	Redevelopment of software portion of system

Table B1. Continued.

Type	Cluster SI	D	SP	BR	Activity
Needs specifics	—	—	✓	—	Refactoring (what was the result?)
Reformative	—	✓	—	—	Reformating documentation but not the code
Reformative	—	✓	—	—	Reformulating the non-code documentation
Evaluative	✓	—	—	—	Regression testing
Performance	—	—	✓	—	Reliability change
Reductive	—	—	—	✓	Removing component or algorithm implementing functionality
Enhancive	—	—	—	✓	Replacing component or algorithm implementing functionality
Updative	—	✓	—	—	Replacing obsolete non-code documentation content
Consultive	✓	—	—	—	Reporting to managers or stakeholders about the software
Adaptive	—	—	✓	—	Resource change incorporated or implemented
Reformative	—	✓	—	—	Restyling the non-code documentation
Consultive	✓	—	—	—	Return of system from outsourcer
Needs specifics	—	—	—	—	Reuse insertion in this system (what was the effect?)
Preventive	—	—	✓	—	Reuse, preparing software for, but involving this system
Reformative	—	✓	—	—	Revising the non-code documentation, but not updating it
Updative	—	✓	—	—	Revising the non-code documentation by updating it
Reformative	—	✓	—	—	Revising or preparing training materials re a system
Needs specifics	—	—	—	—	Rewriting software (what code, or non-code documentation?)
Performance	—	—	✓	—	Robustness change
					S
Evaluative	✓	—	—	—	Searching for things in the documentation
Groomative	—	—	✓	—	Security access changed
Enhancive	—	—	—	✓	Security functions added or replaced
Performance	—	—	✓	—	Service availability changed
Evaluative	✓	—	—	—	Simulating operating conditions, as when testing
Evaluative	✓	—	—	—	Software comprehension
Reformative	—	✓	—	—	SOP manual change, adapting non-code documentation to
Groomative	—	—	✓	—	SOP manual change, code styling
Performance	—	—	✓	—	Speed of execution changed
Performance	—	—	✓	—	Storage utilization changed
Evaluative	✓	—	—	—	Stress testing
Evaluative	✓	—	—	—	Studying the software, system, product, work order, etc.
					T
Training	✓	—	—	—	Teaching courses from existing materials
Adaptive	—	—	✓	—	Technology change incorporated or implemented
Needs specifics	—	—	—	—	Testing done (as a part of what activities?)
Evaluative	✓	—	—	—	Testing, either diagnostic or regression
Evaluative	✓	—	—	—	Testing environment set up or changed
Updative	—	✓	—	—	Test plan incorporation in non-code documentation
Evaluative	✓	—	—	—	Test plan preparation for regression or diagnostic testing
Training	✓	—	—	—	Training done or received
Reformative	—	✓	—	—	Training materials, preparation, or revision
Evaluative	✓	—	—	—	Transfer of system, receiving from development

Table B1. Continued.

Type	Cluster SI	D	SP	BR	Activity
Consultive	✓	—	—	—	Transfer of system to outsourcer
Evaluative	✓	—	—	—	Transition of system, receiving from development
Evaluative	✓	—	—	—	Turnover of system, receiving from development
Consultive	✓	—	—	—	Turnover of system to outsourcer
					U
Needs specifics	—	—	—	—	Unit testing (re properties or business rules?)
Adaptive	—	—	✓	—	Upgrading of platform, hardware or software
Updative	—	✓	—	—	Updating of security documentation to reflect actual situation
Enhancive	—	—	—	✓	Updating of security implementation but not personnel access
Groomative	—	—	✓	—	Updating of security re personnel access
Performance	—	—	✓	—	Up time of system changed
Performance	—	—	✓	—	Use mode changed (e.g., to multi-user)
					V
Groomative	—	—	✓	—	Visibility changed into software
					W
Corrective	—	—	—	✓	Work around created
Consultive	✓	—	—	—	Work around explained or distributed
Needs specifics	—	—	—	—	Write code (as a part of what activities?)
Needs specifics	—	—	—	—	Write documentation (as a part of what activities?)
					X
Needs specifics	—	—	—	—	XP (extreme) software being changed (what done or how?)
					Y
Preventive	—	—	✓	—	Y2K work that does not change externally seen functionality
					Z
Groomative	—	—	✓	—	Zero-out or wipe clean of memory work area

ACKNOWLEDGEMENTS

This paper is a revision, extension, elaboration and expansion of a paper by one of the authors [6] published by the IEEE Computer Society Press. We thank the reviewers for their stimulating comments.

REFERENCES

1. Chapin N. The job of software maintenance. *Proceedings Conference on Software Maintenance–1987*. IEEE Computer Society Press: Los Alamitos CA, 1987; 4–12.
2. Parikh G. The several worlds of software maintenance—a proposed software maintenance taxonomy. *ACM SIGSOFT Software Engineering Notes* 1987; **12**(2):51–53.

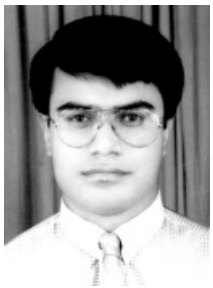
3. Kitchenham BA, Travassos GH, Mayrhauser Av, Niessink F, Schneidewind NF, Singer J, Takada S, Vehvilainen R, Yang H. Toward an ontology of software maintenance. *Journal of Software Maintenance* 1999; **11**(6):365–389.
4. Drucker PF. *Management*. Harper & Row, Publishers: New York NY, 1974; 464–516.
5. ICSM. Empirical studies and management sessions. *Proceedings International Conference on Software Maintenance*. IEEE Computer Society Press: Los Alamitos CA, 2000; 117–276.
6. Chapin N. Software maintenance types—a fresh view. *Proceedings International Conference on Software Maintenance*. IEEE Computer Society Press: Los Alamitos CA, 2000; 247–252.
7. Canning RG. That maintenance 'iceberg'. *EDP Analyzer* 1972; **10**(10):1–14.
8. Swanson EB. The dimensions of maintenance. *Proceedings 2nd International Conference on Software Engineering*. IEEE Computer Society: Long Beach CA, 1976; 492–497.
9. Chapin N. Do we know what preventive maintenance is? *Proceedings International Conference on Software Maintenance*. IEEE Computer Society Press: Los Alamitos CA, 2000; 15–17.
10. Lientz BP, Swanson EB. *Software Maintenance Management*. Addison-Wesley Publishing Co.: Reading MA, 1980; 214 pp.
11. Halpern M. The evolution of the programming system. *Datamation* 1964; **10**(7):51–53.
12. ISPSE. *Proceedings of the International Symposium on Principles of Software Evolution, ISPSE 2000*. IEEE Computer Society Press: Los Alamitos CA, 2001; 332 pp.
13. Swanson EB, Chapin N. Interview with E. Burton Swanson. *Journal of Software Maintenance* 1995; **7**(5):303–315.
14. IEEE. *IEEE Standard Glossary of Software Engineering Terminology*. Institute of Electrical and Electronics Engineers: New York NY, 1990: 83 pp.
15. Kemerer CF, Slaughter SA. Determinants of software maintenance profiles: an empirical investigation. *Journal of Software Maintenance* 1997; **9**(4):235–251.
16. Martin J, McClure CL. *Software Maintenance: The Problem and Its Solution*. Prentice-Hall, Inc.: Englewood Cliffs NJ, 1983; 472 pp.
17. Parikh G (ed.) *Techniques of Program and System Maintenance* (2nd edn). QED Information Sciences, Inc.: Wellesley MA, 1988; 463 pp.
18. Perry WE. *Managing Systems Maintenance*. Q.E.D. Information Sciences, Inc.: Wellesley MA, 1981; 371 pp.
19. Arthur LJ. *Software Evolution*. John Wiley & Sons, Inc.: New York NY, 1988; 254 pp.
20. Bendifallah S, Scacchi W. Understanding software maintenance work. *IEEE Transactions on Software Engineering* 1987; **SE-13**(3):311–323.
21. Gustafson DA, Melton AC, An KH, Lin I. Software maintenance models. *Technical Report*, Department of Computing and Information Sciences, Kansas State University, Manhattan KS, 1990; 14 pp.
22. Martin RJ, Osborne WM. *Guidance on Software Maintenance* (*NBS Special Publication 500–106*). National Bureau of Standards: Washington DC, 1983; 66 pp.
23. IEEE. *IEEE Standard for Software Maintenance* (*IEEE Std 1219–1998*). Institute for Electrical and Electronic Engineers: New York NY, 1998; 47 pp.
24. ISO/IEC. *Information Technology—Software Life Cycle Processes*, ISO/IEC 12207. International Standards Organization: Geneva, Switzerland, 1995.
25. Pigoski TM. *Practical Software Maintenance*. John Wiley & Sons, Inc.: New York NY, 1997; 384 pp.
26. Polo M, Piattini M, Ruiz F, Calero C. MANTEMA: A software maintenance methodology based on the ISO/IEC 12207 standard. *Proceedings 4th International Software Engineering Standards Symposium*. IEEE Computer Society Press: Los Alamitos CA, 1999; 76–81.
27. ISO/IEC. *Software Engineering—Software Maintenance*, ISO/IEC FDIS 14764:1999(E). International Standards Organization: Geneva, Switzerland, 1999; 38 pp.
28. Chapin N. Software maintenance: A different view. *AFIPS Conference Proceedings of the 1985 National Computer Conference*, vol. 54. AFIPS Press: Reston VA, 1985; 328–331.
29. Grady RB. Measuring and managing software maintenance. *IEEE Software* 1987; **4**(9):35–45.
30. Reutter J. Maintenance is a management problem and a programmer's opportunity. *AFIPS Conference Proceedings of the 1981 National Computer Conference*, vol. 50. AFIPS Press: Reston VA, 1981; 343–347.
31. Kajko-Mattsson M. Common concept apparatus with corrective software maintenance. *Proceedings International Conference on Software Maintenance*. IEEE Computer Society Press: Los Alamitos CA, 1999; 287–296.
32. Haworth DA, Sharpe S, Hale DP. A framework for software maintenance: A foundation for scientific inquiry. *Journal of Software Maintenance* 1992; **4**(2):105–117.
33. Harjani D-R, Queille J-P. A process model for the maintenance of large space systems software. *Proceedings Conference on Software Maintenance*. IEEE Computer Society Press: Los Alamitos CA, 1992; 127–136.
34. Lehman MM, Belady LA. *Program Evolution: The Process of Software Change*. Academic Press: New York NY, 1985; 560 pp.
35. Corbi TA. Programming understanding: Challenge for the 1990s. *IBM System Journal* 1989; **28**(2):294–306.

36. Chapin N. Software maintenance life cycle. *Proceedings Conference on Software Maintenance—1988*. IEEE Computer Society Press: Los Alamitos CA, 1988; 6–13.
37. Lehman MM. Programs, cities, students, limits to growth? *Imperial College of Science and Technology Inaugural Lecture Series* 1970; **9**:211–229. Reprinted in [34, ch. 7].
38. Votta LG. Is software worth preserving? The future through the past (keynote presentation). *International Conference on Software Maintenance*, 2000. http://www.brincos.com [10 November 2000].
39. ICSM. Panel 2: preventive maintenance! Do we know what it is? *Proceedings International Conference on Software Maintenance*. IEEE Computer Society Press: Los Alamitos CA, 2000; 11–19.
40. Kajko-Mattsson M. Taxonomy of problem management activities. *Proceedings 5th European Conference on Software Maintenance and Reengineering*. IEEE Computer Society Press: Los Alamitos CA, 2001; 1–10.
41. Couch RF. Evolution of a toll MIS—Bell Canada. *Management Information Systems: Selected Papers from MIS Copenhagen 70—An IAG Conference*, Goldberg W, Nielsen TH, Johnsen E, Josefsen H (eds.) Auerbach Publishers Inc.: Princeton NJ (Studentlitteratur, Sweden), 1971; 163–188.
42. Hsi I, Potts C. Studying the evolution and enhancement of software features. *Proceedings International Conference on Software Maintenance*. IEEE Computer Society Press: Los Alamitos CA, 2000; 143–151.
43. Lehman MM. Rules and tools for software evolution planning and management. *Preprints of FEAST 2000 International Workshop on Feedback and Evolution in Software and Business Processes*. Imperial College, London, 2000; 53–68. http://www.doc.ic.ac.uk/˜mml/f2000 [27 December 2000].
44. Burd E, Munro M. An initial approach toward measuring and characterising software evolution. *Proceedings 6th Working Conference on Reverse Engineering, WCRE'99*. IEEE Computer Society Press: Los Alamitos CA, 1999; 168–174.
45. Tomer A, Schach SR. The evolution tree: A maintenance-oriented software development model. *Proceedings 4th European Conference on Software Maintenance and Reengineering*. IEEE Computer Society Press: Los Alamitos CA, 2000; 209–214.
46. Lehman MM, Ramil RF, Kahen G. Evolution as a noun and evolution as a verb. *Workshop on Software and Organisation Co-evolution*. Imperial College, London, UK. http://www.doc.ic.ac.uk/˜mml/feast [27 December 2000].
47. Bennett KH, Rajlich VT. Software maintenance and evolution: a roadmap. *Proceedings of the 22nd International Conference on Software Engineering*. ACM Press: New York NY, 2000; 75–87.
48. Leeds HD, Weinberg GM. *Computer Programming Fundamentals*. McGraw-Hill Book Co.: New York NY, 1961; 384 pp.
49. Brandon DH. *Management Standards for Data Processing*. D. Van Nostrand Co., Inc.: Princeton NJ, 1963; 143–148.
50. Chapin N. Software maintenance characteristics and effective management. *Journal of Software Maintenance* 1993; **5**(2):91–100.
51. Rosca D, Greenspan S, Feblowlitz M, Wild C. A decision making methodology in support of the business rules lifecycle. *Proceedings 3rd IEEE International Symposium on Requirements Engineering (RE'97)*. IEEE Computer Society Press: Los Alamitos CA, 1997; 236–246.
52. Anderson E, Bradley M, Brinko R. Use case and business rules. *Addendum to the 1997 ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. ACM Press: New York NY, 1997; 85–87.
53. Rosen S. Software. *Encyclopedia of Computer Science*. Van Nostrand Reinhold: New York NY, 1992; 1214–1216.
54. McClure CL. *Managing Software Development and Maintenance*. Van Nostrand Reinhold: New York NY, 1981; Part 2.
55. Huang H, Tsai W-T, Bhattacharya S, Chen XP, Wang Y, Sun J. Business rule extraction techniques for COBOL programs. *Journal of Software Maintenance* 1998; **10**(1):3–35.

AUTHORS' BIOGRAPHIES

Ned Chapin is an Information Systems Consultant with InfoSci Inc. in Menlo Park in California. His decades of experience include all phases of the software life cycle and cover industrial, business, financial, non-profit and governmental organizations. He has also worked in roles from Lecturer to Professor of Information Systems at various universities. Ned's interests span a wide range including software maintenance and evolution, database technology, systems analysis and design, and software management. He is a Registered Professional Engineer, and a Certified Information Systems Auditor. His MBA is from the Graduate School of Business of the University of Chicago, and his PhD is from Illinois Institute of Technology. His e-mail is NedChapin@acm.org

Joanne E. Hale is an Assistant Professor of Management Information Systems at The University of Alabama. Her research interests include software maintenance, software cost estimation, and component-based development and reuse. The National Science Foundation, ExxonMobil, and Texas Instruments have supported her research efforts, and her research has been published in the *Journal of Software Maintenance*, *IEEE Transactions on Software Engineering*, *IEEE Software*, *Journal of Management Information Systems and IEEE Transactions on Systems, Man, and Cybernetics*. She earned her PhD in Management Information Systems from Texas Tech University, and her MA in Statistics and BS in Engineering from the University of Missouri. Her e-mail is jhale@cba.ua.edu



Khaled Md. Khan is a Lecturer in the School of Computing and Information Technology at the University of Western Sydney in Australia. During the past ten years, he has taught computer science at various universities in Europe, Africa, and Asia. Khaled's major research interests include software maintenance processes, characterization of security properties of software components, software metrics, and software quality. His undergraduate and postgraduate degrees are in Computer Science and Informatics from the University of Trondheim in Norway. His e-mail is Md.Khan@acm.org

Juan F. Ramil is a Research Associate and a PhD candidate in the Department of Computing at Imperial College in London. Since 1996, he has been working on the FEAST (Feedback, Evolution And Software Technology) projects. Previously, he worked for nine years in the petroleum industry on the design and implementation of plant automation projects in Venezuela. Juan's research interests include the management of software maintenance and evolution, and effort estimation in the evolution context. Since 1997, he has authored or co-authored 30 papers. He holds an *Ingeniero* degree *Cum Laude* in Electronics and a *Magister* in Business Engineering, both from Simón Bolívar University in Caracas, Venezuela. His e-mail is ramil@doc.ic.ac.uk

Wui-Gee Tan is a Program Manager with the Institute of Systems Science at the National University of Singapore. He returned to academia in 1986 after working for more than 13 years in industry in various information technology roles. His teaching and research interests include software maintenance management and software project management. Wui-Gee received his BSc(Hons), GradDip in Business Administration, and MSc (Computer Science) from the National University of Singapore, and his PhD from the Queensland University of Technology in Australia. His e-mail is wuigee@iss.nus.edu.sg