Purposes, Results, and Types of Software Post Life Cycle Changes

Seokha Koh* · Man Pil Han**

Abstract

This paper addresses the issue how the total life cycle cost may be minimized and how the cost should be allocated to the acquirer and developer. This paper differentiates post life cycle change (PLCC) endeavors from PLCC activities, rigorously classifies PLCC endeavors according to the result of PLCC endeavors, and rigorously defines the life cycle cost of a software product. This paper reviews classical definitions of software 'maintenance' types and proposes a new typology of PLCC activities too. The proposed classification schemes are exhaustive and mutually exclusive, and provide a new paradigm to review existing literatures regarding software cost estimation, software 'maintenance,' software evolution, and software architecture from a new perspective.

This paper argues that the long-term interest of the acquirer is not protected properly because warranty period is typically too short and because the main concern of warranty service is given to removing the defects detected easily. Based on the observation that defects are caused solely by errors the developer has committed for software while defects are often induced by using for hardware (so, this paper cautiously proposes not to use the term 'maintenance' at all for software), this paper argues that the cost to remove defects should not be borne by the acquirer for software.

Keywords : Software Maintenance, Software Warranty Service, Software Evolution, Architecture Erosion, Life Cycle Cost

Received: 2015. 06. 08. Final Acceptance: 2015. 09. 19.

※ This work was supported by the research grant of Chungbuk National University in 2013.

* Corresponding Author, Department of Management Information Systems, Chungbuk National University, e-mail: shkoh@cbnu.ac.kr

** Department of Management Information Systems, Chungbuk National University, e-mail: hau7070@hanmail.net

1. Introduction

In the early 1980s, software evolution literatures introduced the so-called SPE classification scheme classifying software programs into three classes and declared that the laws of software evolution refer only to E-type software [Herraiz et al., 2013]:

- S-type: "Specified programs are derivable from a static specification and can be formally proven as correct or not",
- P-type: "Problem-solving programs attempt to solve problems that can be formulated formally, but which are not computationally affordable. Therefore, the program must be based on heuristics or approximations to the theoretical problem",
- E-type: "Evolutionary programs are reflections of human processes or of a part of the real world. These kinds of programs try to solve an activity that somehow involves people or the real world."

The software life cycle can be divided into two major parts: before and after delivery. The life cycle of a software system after the initial development or the initial delivery release is frequently called post life cycle. An E-type software system evolves as the result of a series of changes during its post life cycle (PLCC).[1] [Belady and Lehman, 1976; Lehman, 1974, 1985, 1996; Lehman and Belady, 1985; Lehman and Rami, 2003; Riaz et al., 2009]. This paper addresses the issues regarding changing an E-type software system after its delivery.

Changing a software system after its delivery is typically called 'maintenance.' Oxford online dictionary defines maintenance as "the process of preserving a condition or situation of being preserved." Sharing the same term 'maintenance, people (including managers) frequently use analogy of hardware maintenance when they address issues regarding software maintenance. The term maintenance, however, bears a significantly different meaning for software as compared with facility or equipment [Hatton, 2007].

NF EN 13306 [2001] defines maintenance, irrespective of the type of items considered except software, as "combination of all technical, administrative and managerial actions during the life cycle of an item intended to retain it in, or restore it to, a state in which it can perform the required function." On the other hand, the typical definitions of maintenance of software are as follows:

- Boehm [1981]: "Modifying of existing operational software while leaving its primary functions intact."
- GAO (The American General Accounting Office): "All work done on a system after it first went into operation or production" [Martin and Osborne, 1983].
- ANSI/IEEE Std. 729 [1983]: "Modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment."

1) In this paper, the change or changes made on a software product after its delivery will be referred as PLCC (post-life cycle change/changes).

- IEEE Std. 1219-1998 [1998]: “Modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment.
- ISO/IEC 14764 [2006]: “The totality of activities required to provide cost-effective support to a software system”, including both pre-delivery maintenance activities (the activities performed during the pre-delivery stage, including planning for post-delivery operations, supportability, and logistics determination) and post-delivery maintenance activities (the activities performed during the post-delivery stage, including software modification, training, and operating a help desk). “The maintenance process contains the activities and tasks necessary to modify an existing software product while preserving its integrity and is initiated by a modification request (MR)”

GAO includes explicitly all changes made in post life cycle (PLCC) into the single category ‘maintenance.’ For the accountants this was clear division: By definition, every project has a cut-off date, the date of initial delivery and all costs prior to that date are charged to the development project, all cost due to changes after that date are charged to maintenance operation [Sneed, 2004]. The definition of ANSI/IEEE Std. 729 1983 is almost the same as that of GAO in the essence and these two definitions are still generally accepted as the standard definitions of software maintenance. Other definitions may be interpreted to include implicitly all kinds of PLCC into the single category ‘maintenance.’

In short, for a hardware product, maintenance generally means preserving and restoring of its original state. That is, greasing or doing something to prevent it from wearing out or deteriorating physically or restoring broken or deteriorated parts to perform the required function again. Software, however, neither wears out nor deteriorates physically and it is needless to grease software as well as impossible. So, for a software system, the term ‘maintenance’ is used to refer to changing it from its original state [Koh, 2014].

A software system, unlike hardware, may grow much bigger after its first delivery. Belady and Lehman [1976] classify software products ‘static’ or ‘dynamic’ according to their growth rate. Software evolution deals with the problems associated with dynamically growing software products [Sneed, 2004]. Software evolution, like software ‘maintenance’, is concerned with what happens to software programs after the initial release and deals with the process by which software programs are modified and adapted to their changing environment [Herraiz et al., 2013]. In the context of software evolution, the focus is on system growth and improvement activities like error correction are generally not considered relevant [Sneed, 2004]. Adopting this point of view, Koh [2014, 2015] classified PLCC activities into two categories of ‘maintenance’ and ‘augmentation’ according to their impact on growth rate.

Koh [2014, 2015] divides the level of PLCC work into PLCC endeavor and PLCC activity, rigorously classifies PLCC endeavors according to the result of PLCC endeavors, rigorously defines the life cycle cost of a software product, reviews classical definitions of software 'maintenance' types, and proposes a new typology of PLCC activities. This paper elaborates the works of Koh [2014, 2015]. The result will provide a new paradigm to review existing literatures regarding software cost estimation, software 'maintenance,' software evolution, and software architecture erosion from a new perspective. This paper addresses the issue of how to protect the long-term interest of the software acquirer properly too, as well as short-term interest.

2. Hierarchy of PLCC Work

In this paper, we will define a generic level of PLCC work as follow: [2)]

- PLCC endeavor: In outsourcing environment, it includes all kinds of PLCC work contracted separately under split contract and the PLCC work initiated by a modification request (MR) or a change request of the acquirer.

In management information system environment or 'in-house' development environment in which a software product is produced by an enterprise to support of its own business and administrative operations [Jones, 2000], the PLCC endeavor includes all kinds of PLCC work executed and managed as a separate unit of work, corresponding to those in outsourcing environment.

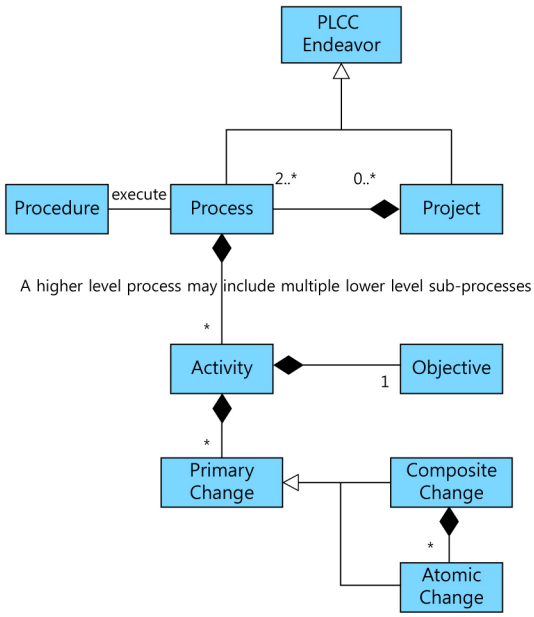
According to ISO/IEC 14764-2006, software maintenance is the 'modification' initiated by a MR and is differentiated from migration which involves PLCC too. It defines MR as the "generic term used to identify proposed modifications to a software product that is being maintained." According to ISO/IEC 14764-2006,

- During process implementation, the maintainer should "establish the plans and procedures which are to be executed during the maintenance process."
- During problem and modification analysis, "the maintainer analyzes MRs/PRs; replicates or verify the problem; develops options for implementing the modification; documents the MR/PR, the results, and execution options; and obtains approval for the selected modification option."
- During modification implementation, "the maintainer develops and tests the modification of the software product",
- Maintenance review/acceptance "ensures that the modification to the system are correct and that they were accomplished in accordance with the approved standards using the correct methodology",
- "During a system's life, it may have to be modified to run in different environments, In order to migrate a system to a new envi-

2) This paper is the revision of the conference paper Koh [2015]. This section includes Sec. 4. Hierarchy of PLCC Work of the paper and a part of Sec. 5 Software Maintenance and Other Types of Software Post Life Cycle Changes of Koh [2015], unchanged.

ronment, the maintainer needs to determine the actions needed to accomplish the migration, and then develop and document the steps required to effect the migration", Migration includes, "beside adaptation of a software product to the changed environment, conversion of data, parallel operation of both old and new products, user training, etc."

- Once a software product has reached the end of its useful life, it should be retired. "An analysis should be performed to assist in making the decision to retire a software product. The analysis is often economic-based and may be included in the retirement plan, and should if it is cost effective."



Source: Koh [2015].

〈Figure 1〉 Hierarchy of PLCC Work

A PLCC endeavor may be managed as either a process or a project composed with multiple processes (refer <Figure 1>). PMI [2013] defines processes, procedures, projects, and activities as:

- Process: "A systematic series of activities towards causing an end result such that one or more inputs will be acted to create one or more outputs."
- Procedure: "An established method of accomplishing a consistent performance or result, a procedure typically can be described as the sequence of steps that will be used to execute a process."
- Project: "A temporary endeavor undertaken to create a unique product, service, or result."
- Activity: "A distinct, scheduled portion of work performed during the course of a project."

PLCC endeavor includes process and project as its sub-categories. A project is composed with multiple processes. On the other hand, a process may belong to no project and be managed as an independent and separate work unit. A process is composed with multiple activities and may include multiple sub-processes. OMG's [2011] definitions of process and activity coincide with those of PMI [2013]. An activity has a unique objective, whose achievement can be assessed objectively. An activity may be composed with multiple primary changes. The category of primary change includes two sub-categories: composite change and atomic change. Regarding an object-oriented software system as a directed graph of arbitrary artifacts, Lehnert et al. [2012] classifies individual software changes into two categories as followings and we will adopt and extend Lehnert et al.'s definitions of atomic change and composite

change for general software systems other than an object-oriented software system without any further definition:

- Atomic changes: correspond to elementary tree edit operations of add node, delete node, delete edge, and update property,
- Composite changes: consist of sequences of atomic changes and are based on previous research on regression testing and traceability maintenance.
  - ✓ Move: move one sub-graph to another node,
  - ✓ Replace: replace one sub-graph by another sub-graph,
  - ✓ Split: split one sub-graph into several sub-graphs,
  - ✓ Merge: merge several sub-graphs into one,
  - ✓ Swap: exchange two sub-graphs.

In this paper, the activity is regarded to be the basic unit of work which has a unique end or purpose. On the other hand, a process or a project may have multiple ends or purposes.

3. Maintenance and Other Types of PLCC Endeavors

3.1 Maintenance and Major PLCC Endeavors[3)]

Maintenance literatures generally define software maintenance as a special type of software modification. Some authors include only minor changes in 'modification.' For example, Hunt et al. [2008] exclude the following types of changing endeavors from 'modification':

- Major redesign and redevelopment (more than 50% new code) of a new software product performing substantially the same function.
- Design and development of a sizable (more than 20% of the source instructions comprising the existing product) interfacing software package which requires relatively little redesign of the existing product.
- Data processing system operation, data entry and modification of values in the database. On the other hand, modification of the software product's code, documentation, or data base structure is typically included into maintenance.

ISO/IEC 14764-2006 differentiates 'new development' form maintenance to include major PLCC whose amount of the costs and resources do not exceed the fixed price of their initial development. That is, ISO/IEC excludes the PLCC whose size is bigger than some significant ratio of initial development from maintenance. Hatton [2007] used PLCC projects whose duration is under 40 hours in his empirical study on software maintenance. Excluding 'major PLCC' from software maintenance, it typically has following characteristics [Abran and Nguyenkim, 1993]:

- The size and complexity of each maintenance work request are such that one or two resources can usually handle it;

3) This section is the same as the corresponding part of Koh [2015], except the first sentence.

- Maintenance work requests come in more or less randomly, and cannot be accounted for individually in the annual budget-planning process;
- Minor enhancements (adaptive) work requests in the enhancement category are reviewed with customers and can be assigned priorities;
- The maintenance workload is not managed using project management techniques, but rather with queue management techniques;
- Maintenance has a broader scope of configuration management with more operational considerations.

Since MRs come in more or less randomly and cannot be accounted for individually in the annual budget-planning process, the cost of software maintenance work is generally covered by an annual maintenance fee, which is typically a certain percentage of the original development cost or of the purchasing price [Sneed, 2004]. Without any explicit mention, however, users tend to believe that they are given the right to ask for whatever they deem necessary to make their work easier while the party responsible for the maintenance contends that the maintenance fee covers only the costs of error correction and essential adaptations required to keep the system in operation and that all else including optimization, renovations, enhancements and non-essential adaptations should be charged extra [Sneed, 2004]. It seems to be one of the most important causes of the chaos in the literature and practice regarding software maintenance for authors and practitioners to use the term 'software maintenance' without mentioning its scope explicitly [Chaptin et al., 2001; Sneed, 2004]. So, it is necessary to assort software maintenance or PLCC according to the size of necessary work.

3.2 An Exhaustive and Mutually Exclusive Typology of PLCC Endeavors[4)]

As noted above, there is no generally accepted agreement on what software maintenance should be. GAO defines software maintenance to include virtually all kinds of PLCC. That of IEEE seems to include virtually all kinds of PLCC, for there can hardly be other purpose of software 'modification' than correcting error, improving performance or other attributes, or adapting the product to a changed environment. ISO/IEC seems to include only small PLCC requested rather randomly in their 'supporting activities.'

〈Table 1〉 Types of Software PLCC

Reuse Ratio of existing system	Functional Growth Rate of new system	
	low	high
low	Replacement	Retirement
high	Modification	Enlargement

Based upon two parameters of the reuse ratio of existing system and the functional growth rate of resulting system, Koh [2014] classifies PLCC into four types of maintenance, augmentation, replacement, and retirement. In this paper,

4) This section is almost the same as the section 'A Rigorous Typology of Software Post Life Cycle Changes' of Koh [2015]. The main difference is that 'augmentation' is substituted with 'enlargement' in this paper. This section includes a paragraph of 'Maintenance Process and Software Life Cycle' of Koh's [2015] corresponding section 5 too.

we suggest to change the term 'maintenance' and 'augmentation' to 'modification' and 'enlargement', respectively, and not to use the term 'maintenance' at all.

Here, reuse includes any replacement effort saved from redeploying artifacts of an existing software product: specifications, programs, screens, reports, data files, etc. The term 'maintenance' provokes confusions and it seems desirable to refrain from using the term 'maintenance' for software. So, we propose not to use the term 'maintenance' for Software PLCC at all. In <Table 1>, 'modification' virtually replaces 'maintenance.'

When a software product is used no more and/or only a small portion of a software product is reused in a new software product which shares only small portion of functionality with the existing software product, the software product is regarded to be retired. Koh [2014] defines the life cycle of a software product as the period between its inception and retirement, which includes the initial development and all changing activities afterward. 'New development whose amount of the costs and resources do not exceed the initial fixed price' [ISO/IEC 14764-2006] and 'redevelopment' that usually occurs on a new platform or with a different software environment are included in PLCC endeavors as modification, replacement, or enlargement. Decisions regarding these activities affect life cycle cost of a software product system. It is noticeable that this classification scheme classifies PLCC endeavors during the whole life cycle of a software product exhaustively and mutually exclusively.

It is very important for the management to build a company wide software audit to identify what software products are active on the network day by day and to retire a software product productive no longer [Koh, 2014]. According to a survey on a cross section of businesses, about 75% of respondents said that they had no system in place to deal with retiring software products, more than 70% reported that there were redundant, deficient or obsolete software products being changed and supported on their networks, 40% estimated that unwanted software products consumed more than 10% of their budget, 40% reported that their company conducted audits only on an as-needed basis, and just over 13% said that they never conducted software audit at all [Kooser, 2005].

A software product should be retired through the analysis to determine if it is cost effective to [ISO/IEC 14764-2006]:

- Retain outdated technology,
- Shift to new technology by developing a new software product,
- Develop a new software product to achieve modularity,
- Develop a new software product facilitate maintenance;
- Develop a new software product to achieve standardization,
- Develop a new software product to facilitate vendor independence.

4. Purposes of Software Maintenance

4.1 Prevention and Correction

For hardware, prevention and correction are

Purpose/ Type	Key Words				
	NF EN 13306 (Hardware)	Lientz and Swanson[1980]	IEEE Std 1219-1998	ISO/IEC 14764 : 2006	This Paper
Correction	Fault recognition, Required function	Faults	Discovered faults	Discovered problems	Faults
Reactive correction	-	-	-	-	Manifested as failures
Emergency/ Emergent correction	-	-	Unscheduled	Unscheduled, Temporary	Unscheduled, Temporary
Prevention	Reduce the probability of failure or degradation	Forestall or reverse deterioration	-	Latent faults, Before operational	
Perfection	-	Improve	Improve	Latent faults, Before manifested	-
Proactive correction					Latent faults
Enhancement	-	-	-	New requirement	New requirement, Non-Functional
Adaption	-	Suiting to different conditions	Changes of environment	Changes of environment	Changed requirements
Augmentation					New requirements, Functional

Correction
- **NF EN 13306: (Hardware)** Maintenance carried out **after fault recognition** and intended to put an item into a state in which it can **perform a required function**. This includes deferred maintenance as its sub-type.
- **Lientz and Swanson**: Changes are made in order to **remove faults**. This includes emergency fixes and routine debugging.
- **IEEE**: Reactive modification of a software product performed after delivery to **correct discovered faults**. This includes emergency maintenance as its sub-type.
- **ISO/IEC**: The reactive modification of a software product performed after delivery to **correct discovered problems**. This includes emergency maintenance as its sub-type.
- **This paper**: Removing faults. The fault is defined as failing to satisfy requirements and removing faults denotes letting unsatisfied requirements satisfied. This includes adding required but unimplemented functionality. This category is divided further into proactive correction and reactive correction.

Prevention
- **NF EN 13306**: (Hardware) Maintenance carried out at predetermined intervals or according to prescribed criteria and intended **to reduce the probability of failure or the degradation** of the functioning of an item. This includes scheduled/planned, predetermined and condition base d maintenance as its sub-types.
- **Lientz and Swanson**: Changes made in order to **forestall or reverse deterioration**.
- **ISO/IEC**: Modification of a software product after delivery to **detect and correct latent faults** in the software product before they become operational faults.

Proactive Correction
- **This paper: Detecting and correcting latent faults** in a software product before they are manifested as failures.

Reactive Correction
- **This paper: Correcting faults** in a software product after they are manifested as failures. This includes emergent correction its sub-type.

〈Figure 2〉 Definitions of Maintenance Types, or Purposes of PLCC Endeavors

Emergency
- IEEE: Unscheduled modification performed to **keep a system operational**.
- **ISO/IEC: Unscheduled** modification performed to **temporarily keep a system operational** pending corrective maintenance.

Emergent Correction
- **This paper: Temporarily keeping a system operational**, pending reactive maintenance.

Perfection
- **Lientz and Swanson**: Changes are made in order to **improve**. This includes customer enhancements, improvements to documents, and optimization.
- **IEEE:** Modification of a software product after delivery to **improve performance or maintainability**.
- **ISO/IEC**: Modification of a software product after delivery to **detect and correct latent faults** in the software product before they are manifested as failures.

Adaption
- **Lientz and Swanson**: Changes are made in order to become **suited to a different condition**. This includes accommodating changes to data inputs and files and accommodating to hardware and system software.
- **IEEE**: Modification of a software product performed after delivery to keep a computer program **usable in a changed or changing environment**.
- **ISO/IEC**: The modification of a software product, performed after delivery, to keep a software product **usable in a changed or changing environment.**
- **This paper: Satisfying existing but changed requirements**. The modification is typically performed to keep a software product **usable in a changed or changing environment**.

Enhancement
- **ISO/IEC**: Modification to an existing software product to **satisfy a new requirement.**
- **This paper: Satisfying new non-functional requirements.**

Augmentation
- **This paper: Satisfying new functional requirements.**

〈Figure 2〉 Definitions of Maintenance Types, or Purposes of PLCC Endeavors (continued)

traditionally regarded as the major types of maintenance. ISO/IEC 14764 : 2006 defines preventive maintenance and corrective maintenance as the maintenance to reduce the probability of failure or degradation of the functioning of an item and as the maintenance to put an item into a state in which it can perform a required function after a fault is recognized, respectively (refer <Figure 2>).

For software too, Lientz and Swanson [1980] recognize the corrective maintenance as a major type of maintenance and define it as removing faults. For hardware, corrective maintenance consists primarily of removing faults too. For hardware, however, the faults are those generated by usage after the product was released or changed previously while the faults are those has existed already when the product was released or changed for software. So, correction means improving the product for software while it means recovering the original state of the product for hardware. That is, the term correction denotes quite different phenomena for software and hardware. This difference may induce confusions

among managers who are accustomed to hardware maintenance and may produce significant results because it is managers who make important decisions regarding software maintenance.

Lientz and Swanson [1980] recognize the preventive maintenance as a major type of software maintenance too, and define it as forestalling or reversing deterioration. It is notable that their definition of software preventive maintenance is essentially the same as NF EN 13306's definition of hardware preventive maintenance: preventing a product from deterioration or degradation by usage or passage of time. Software, however, never deteriorates by using or by passage of time. So, it is needless to try to prevent a product from deterioration by usage or passage of time. For software, deterioration can occurs only as side effects of deliberate changes.

The effort performed to prevent, detect and correct faults, however, is generally called software quality assurance or verification & validation (V&V) [Air Force Instruction 16-1001, 1996; ANSI/IEEE Std. 729-1983; Lewis, 1992, p. 7; Ratkin, 1997, pp. 51-52; Schulmeyer and MacKenzie, 2000, p. 2]. So, if prevention denotes preventing new errors from being committed during PLCC endeavors, it is not differentiated from software assurance or V&V. So, preventing new errors from being committed during PLCC endeavors should not be regarded as a generic type of independent and distinct PLCC endeavors

So, ISO/IEC 14764-2006 defines software preventive maintenance as finding and correcting 'existing but not-operational-yet' faults, and classifies it as a subtype of correction. Here, however, prevention denotes preventing existing faults from being operational in ISO/IEC 14764-2006 while it denotes preventing new faults from occurring in NF EN 13306. This difference may induce confusions also among managers who are accustomed to hardware maintenance.

In this paper, we define correction as removing faults and classify it into proactive or reactive according to whether the faults to be removed are manifested as failures or not, respectively (refer <Figure 2>). It includes adding required but un-implemented functionality. Our definitions of proactive correction and reactive correction classify fault correction exhaustively and mutually exclusively.

4.2 Adaption, Enhancement and Augmentation

Lientz and Swanson [1980] classified software maintenance into four broad categories of adaptive maintenance, corrective maintenance, perfective maintenance, and others. Others include prevention. Their classification is broadly accepted as a standard classification of software maintenance [Hunt et al., 2008; Koh, 2014]. Following Lientz and Swanson's scheme, IEEE Std. 1219-1998 classifies software maintenance into three broad categories of adaptive, corrective, and perfective maintenance. It defines emergency maintenance as a subtype of corrective maintenance.

ISO/IEC 14764-2006 classifies maintenance into two broad categories of correction and enhancement: correction is divided further into corrective maintenance and preventive maintenance, and enhancement encompasses all types

of changing effort other than fault correction. According to it, modification requests are to be classified as a correction or enhancement and later identified as corrective, preventive, adaptive, or perfective. Adaptive maintenance is 'the modification of a software product, performed after delivery, to keep a software product usable in a changed or changing environment' and perfective maintenance is 'modification of a software product after delivery to detect and correct latent faults in the software product before they are manifested as failures.'

ISO/IEC notes that adaptive maintenance provides 'enhancement necessary to accommodate changes in the environment in which a software product operate' and that perfective maintenance provides 'enhancement for users, improvement of program documentation, and recording to improve software performance, maintainability, or other attribute.' ISO/IEC's note for perfective maintenance coincides with Lientz and Swanson's [1980] or IEEE Std. 1219-1998's definitions of perfective maintenance. According to its definition, however, ISO/IEC's perfective maintenance falls under the category of ours proactive correction. The discordance between the definition and note for ISO/IEC's perfective maintenance may produce confusion. Moreover, what is being perfect? We strongly suggest refraining from using the term 'perfect.'

In this paper, we classify the purpose of software maintenance into four broad categories: correction, adaption, enhancement, and augmentation (refer <Figure 2>). Adaptation, enhancement, augmentation denote satisfying existing but changed requirements, new nonfunctional requirements, and new functional requirements, respectively. We assume that no one changes an existing software product to deteriorate it, although it may be deteriorated unexpectedly by being changed. It is noticeable that our definitions classify the purpose of software PLCC activities exhaustively and mutually exclusively.

<Table 2> shows the distribution of time spent among 'maintenance' types. In the table, maintenance types are classified and named according to one of existing classification schemes respectively by authors. The table shows the distribution of time spent among 'maintenance' types varies widely among authors. Hatton [2007] interprets this result as partly due to the fact that many 'maintenance'activities are difficult to classify according to these definitions. One of the difficulties is, according to Hatton [2007], that there is considerable overlap among perfective, corrective, and adaptive maintenance tasks, and it is not unusual, for example, while performing adaptive maintenance to find a defect, or perhaps decide that some perfective rewriting is necessary to add a new feature [Hatton, 2007]. Hatton's [2007] observation notes that a 'maintenance' endeavor may have and frequently has multiple purposes.

<Table 2> also shows, regardless of the meaning of prevention, that empirical researches reveal that software maintenance named preventive is very rare in practice. It manifests clearly that prevention is not an important generic purpose of software maintenance.

Sneed [2004] also argues that high percentage of perfective maintenance may be interpreted so that a lot of development is charged to 'mainten-

〈Table 2〉 Percentages of 'Maintenance' Efforts by Types

Author		LS80	HW84	DK92	YLC94	GL96	SN96	KS97	HT07
Origin of Data		USA	-	-	Hong Kong	-	-	-	Britain
Types of Change	Enhance	**51.3**	**28**	**25**	**39.7**	**23**	**35**	**5**	**40**
	Customer enhancements	41.8	-	-		-	-	-	-
	Improvements to documents	5.5	-	-		-	-	-	-
	Optimization	4.0	-	-		-	-	-	-
	Adapt	**23.6**	**29**	**46**	**9.8**	**42**	**52**	**83**	**54**
	Data inputs and files	17.4	-	-		-	-	-	-
	HW and system SW	6.2	-	-		-	-	-	-
	Correct	**21.7**	**19**	**18**	**15.7**	**37**	**9**	**12**	**6**
	Emergency fixes	12.4	-	-		-	-	-	-
	Routine debugging	9.3	-	-		-	-	-	-
	Others	**3.4**	**24**	**11**	**25.8**	-	**4**	-	-
	Prevention	3.4	-	-		-	-	-	-
	Answering questions	-	-	11	12.7	-	-	-	-
	Documentation	-	-	-	9.0	-	-	-	-
	Tuning	-	-	-	7.1	-	-	-	-
	Re-engineering	-	-	-	6.0	-	-	-	-

Legend: DK02 = [Dekleva, 1992], HT07 = [Hatton, 2007], GL = [Glass, 1996], HW84 = [Helms and Weiss, 1984], KS97 = [Kemerer and Slaughter, 1997], LS80 = [Lientz and Swanson, 1980], SN96 = [Sneed, 1996], YLC94 = [Yip et al., 1994].
Source: Koh [2015].

ance' budget. That is, in evolutionary or iterative development environment, especially where the same group performs both development and maintenance, development of new increments may be easily classified as perfective maintenance. Hatton [2007] argues that this problem is aggravated by the fact that a change of a type often includes or induces changes of other types or that initial appraisal of the type of change necessary is often inaccurate to result in other type of change implemented. It is notable that Yip et al. [1994] separate out software re-engineering from perfective maintenance into another type.

Correction, adaptation, enhancement, and augmentation are purposes of PLCC endeavors. The purpose cannot be an appropriate standard of the classification of PLCC endeavors, because a PLCC endeavor may have multiple purposes. That is, classification of PLCC endeavor according to their purposes does not classify PLCC endeavors mutually exclusively. The classification overlaps. Moreover, the purpose may change during a PLCC endeavor.

Because there are considerable overlap and transitions among various kind of maintenance tasks, prediction of the effort necessary for any kind of change across a range of software products is generally very inaccurate at start and become considerably more accurate as time goes by

[Hatton, 2007]. Empirical evidence seems to imply that experience does not improve maintainer's estimation: that is, one could not, except for corrective, small and simple maintenance tasks, have more confidence in the predictions of an experienced maintainer than the predictions of an inexperienced maintainer [Jorgensen et al., 2000].

5. Results of PLCC Endeavors

5.1 Deterioration of Software

The results of a PLCC endeavor may be different from its original purposes. No one would change a software system to deteriorate it. As a side effect of a PLCC endeavor, however, new faults may be introduced, making the changed software system deteriorate. That is, although deterioration cannot be the purpose software PLCC, it can be the result of a software PLCC endeavor.

Another type, much more important, of deterioration that can be caused by PLCC endeavors is erosion of architecture. The deterioration or erosion of software architecture is addressed using various terms such as architectural decay, architecture degeneration, architecture drift, architecture erosion, code decay, design erosion, software aging, software erosion, or software entropy [Dalgarno, 2009; de Silva and Balasubramaniam, 2012; Eick et al., 2001; Grottkeeet et al., 2008; Hochstein and Lindvall, 2005; Huang et al., 1995; Izurieta and Bieman, 2007; Jacobson, 1992; Koh, 2012; Parnas, 1992; Perry and Wolf, 1992; Riaz et al., 2009; Stringfellow et al., 2006; van Gurp and Bosch, 2002]. Although these terms imply that erosion occurs at different abstraction levels, the underlying perspective in each discussion is that software erosion is a consequence of changes that violate design principles [de Silva and Balasubramaniam, 2012].

The architecture erosion of a software product results from either violating architectural principles or insensitivity to the architecture, and makes the software more complex, harder to understand, and harder to change, and ultimately makes it become progressively less satisfactory in use [de Silva and Balasubramaniam, 2012; Perry and Wolf, 1992]. Eroded architecture can be repaired or recovered [Bellay and Gall, 1997; de Silva and Balasubramaniam, 2012; Gannod and Cheng, 1999; Harris et al., 1995]. Fowler's [1999] refactoring may be considered as an example of architecture recovery methods. Architecture erosion may be prevented by using architecture conscious PLCC processes too [Koh, 2013].

5.2 Evolution of Software

As the result of a sequence of PLCC, a software product evolves. Lehman published the first version of the laws of software evolution in 1974, revised them in 1978 and 1980s, and published the last version of the laws of 'E-type' software evolution in 1996. The laws of software evolution have remained unchanged in essence thereafter [Herraiz et al., 2013]. <Table 3> shows the 1996 version of the laws and their empirical evidence [Herraiz et al., 2013; Koh, 2014].

Laws of software evolution assert that an E-type system either becomes progressively less satisfactory in use or must be continually

adapted, that the adaptation makes the system become progressively bigger, more complex, and more difficult to comprehend and change it, and that rigorous work should be done to maintain or reduce complexity. <Table 3> shows, however, only the laws of continuous change and continuing growth are consistently validated by empirical studies. That is, an E-type system becomes progressively less satisfactory in use unless it is not continually adapted and augmented with new functional capability.

The rest of the laws are not verified. That is, some systems remain relatively constant in their complexity after a sequence of PLCC even if

〈Table 3〉 Laws of Software Evolution and their Empirical Evidences

Law of Software Evolution	Empirical Study on the Hypotheses of 1990s								
	GT 2000	GT 2001	BP 2003	Ro 2005	He 2006	Ko 2007	IF 2010	Ne 2013	Va 2010
Law of Continuous Change An E-type system must be continually adapted, or else it becomes progressively less satisfactory in use.	V	V	V	V	V	V	V	V	V
Law of Increasing Complexity As an E-type is changed, its complexity increases and becomes more difficult to evolve unless work is done to maintain or reduce complexity.	I	I	V	I	I	I	I	I	I
Law of Self-Regulation Global E-type system evolution is feedback regulated.	I	I	V	I	I	-	P_v	I	V
Law of Conservation of Organizational Stability The work rate of an organization evolving an E-type software system tends to be constant over the operational lifetime of that system or phases of that lifetime.	I	I	V	I	I	P_i	V	V	I
Law of Conservation of Familiarity In general, the incremental growth (growth rate trend) of E-type system is constrained by the need to maintain familiarity.	I	I	V	I	I	-	P_v P_i	-	V
Law of Continuing Growth Functional capability of E-type systems must be continuously enhanced to maintain user satisfaction over system lifetime.	V	V	V	V	V	V	V	V	V
Law of Declining Quality Unless rigorously adapted and evolved to take into account changes in the operational environment, the quality of an E-type system will appear to be declining.	-	-	V	-	-	-	I	I	I
Law of Feedback System E-type evolution processes are multi-level, multi-loop, multi-agent feedback systems.	-	-	-	-	-	-	P_v	I	-

Legend: V = validated, I = invalidated, Pv = partially validated, Pi = partially invalidated, GT2000 = [Godfrey and Tu, 2000], GT2001 = [Godfrey and Tu, 2001], BP2003 = [Baouer and Pizka, 2003], Ro2005 = [Robles et al., 2005], He2006 = [Herraiz et al., 2006], Ko2007 = [Koch, 2005, 2007], IF2010 = [Israeli and Feitelson, 2010], Ne2013 = [Neamtiu et al., 2013], VA2010 = [Vasa, 2010].

rigorous efforts to maintain or reduce complexity are not devoted deliberately during the PLCC processes.

5.3 ASR: Architecturally Significant Requirements

ASRs are typically defined to be those requirements that have measurable impact on the architecture of a software system [Chen et al., 2013]. Dependencies between requirements and architecture are reciprocal: ASRs play a significant role in shaping the architectural design of a system and constraining the set of viable architectural alternatives and the existing architecture constrains the financial viability of implementing new feature requests [Cleland-Huang et al., 2013].

In software engineering, nonfunctional requirements describe how well functions of a software system are accomplished while functional requirements describe what a system does, and they are regarded to represent global concerns on the development and operational costs of a system [Niu et al., 2013]. Therefore, nonfunctional requirement attributes such as maintainability, customizability, flexibility, reusability, extensibility, adaptability, accuracy, reliability, availability, safety, dependability, interoperability, composability, performance, efficiency, etc., are generally regarded as natural candidates of ASRs [Mirakhorli, 2011; Niu et al., 2013].

Such nonfunctional requirements exhibit a wide reaching impact throughout the entire software lifecycle including development [Cleland-Huang et al., 2013; Mirakhorli, 2011]. They, however, seldom cause and drive PLCC. So, in the context of evolution, architecture is the underlying factor and ASRs are indicators which the quality of architecture is loaded on as the underlying factor. That is, nonfunctional requirements become architecturally important when and only when their attributes are deteriorated by the deteriorated architecture. Nonfunctional ARSs are architecturally important only as indicators of the quality of a system's architecture.

It is the functional requirements changed or newly introduced that generally cause and drive the evolution of a software system and that let architectural decisions to be made. Nonfunctional ARSs generally constrain the set of viable architectural alternatives driven to fulfill the functional requirements. The functional requirements which drive architecturally important decisions are very specific to the evolution history of a software system. We propose to denote such architecturally important functional requirements specific to software system as system-specific ARS and denote nonfunctional ARSs as generic ARS. It is important very much to differentiate the two kinds of ARSs. In practice, the main focus of architectural consideration should be given to the specific ARSs.

6. Discussions: Who should Bear the Cost of PLCC Activities?

It seems natural and proper for the acquirer bear the cost to adapt, enhance or augment its software system. Who, however, should bear the cost to remove faults? The answer is simple: The one who has caused the faults.

For both hardware and software, the cost to remove the faults which occurred after the warranty period is generally borne by the acquirer. For hardware, it is proper to regard that such faults are caused by using it. So, it is also proper for the user to bear the cost to remove such faults. For software, it is errors which have existed in a system since its delivery that causes faults, and the acquirer is seldom responsible for the errors. So, it is not the acquirer who should bear the cost to remove faults of a software system, regardless when the faults are manifested or become operational. Then, who? It is the developer or one of the maintainers, if any, who changed the system previously.

ISO/IEC 14764 [2006, pp. 25-28] recommends for the acquirer to agree on 'maintenance' models with the original developer or a third party 'maintainer', which address all types of 'maintenance' and include new development unless the amount of the costs and resources exceed the initial fixed price. It recommends two types of comprehensive 'maintenance' contract with a fixed price:

- Blanket contract with fixed amounts: It includes all types of 'maintenance' and may include new development.
- Split contract: It typically includes 'corrective maintenance' (reactive correction) for an agreed period. Preventive, perfective (proactive correction), and adaptive 'maintenance' are usually contracted separately for each.

If the developer is designated as the 'maintainer in advance, according to ISO/IEC 14764-2006, the developer receives additional monetary reward for removing the faults it has committed regardless contract type. The developer may interpret the reward as an incentive for making defects and leaving it un-removed over warranty period, although with blank contract with fixed amount it is not sure how much the reward will be. Even if a third party is designated as the 'maintainer,' the developer does not receive any penalty for the faults left over warranty period. So, the developer has no reason to mind leaving faults un-removed over warranty period.

So, each error correcting activity should be separated from other activities in a PLCC endeavor, and its cost should be charged to the one responsible for the error. This may be very costly in practice. The academics, however, should provide some guidelines to help acquirers to let the developers be more responsible for their product.

Maintainability is affected by the architecture, design, the coding and its programming language and the testing activities and affects the cost of PLCC endeavors [ISO/IEC 14764, 2006, p. 28]. ISO/IEC 14764-2006 recommends that the acquirer should establish and clarify the maintainability requirements and let the capability to monitor and evaluate maintainability criteria identified for each requirement to be developed during the initial development, especially when maintenance service is provided by a third party and the maintainer cannot be involved in the development process, which is the general case.

However, we argue, without citing concrete empirical evidences, following postulations,

Postulation 1: It is very costly to measure the quality of software systems objectively.

Postulation 2: There is information imbalance about quality of software quality between the acquirer and the developer. That is, the acquirer cannot access to all of the information the developer possesses. So, it is generally not economical for the acquirer to measure the quality of a software system which it owns for itself.

Postulation 3: Defects associated with coding and functional requirements tends to be manifested sooner or later. So, it is possible for the acquirer to motivate the developer to prevent such defects from being delivered by letting the developer bear the cost to remove such defects.

Postulation 4: Non-functional quality attributes including maintainability are harder to measure objectively than functional quality attributes. Defects associated with non-functional requirements are seldom manifested objectively and clearly even after a considerable time has elapsed since the delivery. So, it is virtually impossible for the acquirer to motivate the developer to prevent such defects from being delivered, at least, under current practice.

The postulations above lead to the assertion that the long-term interest of the acquirer is not protected properly [Koh and Ji, 2013] due to short warranty period while main concern of warranty is given to the defects detected easily. ISO/IEC 14764-2006 recommends the acquirer to bear the cost to remove defects after warranty period. This obviously motivates the developer to neglect its basic obligation to increase the quality of products it produces. Ultimately, this produces distrust between the acquirer and developer, hampering healthy development of software industry. Further researches on how to protect the acquirer's long-term interests is seriously needed.

7. Conclusions

For the defects of a hardware product used for a certain period, the user is, at least partially, responsible for the defects because the user caused the product worn out by using it. For the defects of a software system, the user is not responsible for the defects. It is the developer or, if any, third party maintainers who had committed the errors which cause the faults. So, error correction should be distinguished from adapting, enhancing, and augmenting, and the cost of error correction should be charged separated from the costs of adapting, enhancing, and augmenting.

Error correction, however, is often performed with adapting, enhancing, or augmenting. Errors may be detected and corrected during PLCC endeavors which are originally started to adapt, enhance, or augment. So, the cost of error correction can be charged to those responsible for each error only by tracing the costs at the level of activity, not at the level of endeavor. The paradigm proposed in this paper makes it possible, at least theoretically, to trace the costs at the level of activity and separate the cost of activities to correct each error from the cost of

other activities.

Adapting, enhancing, augmenting, and even error-correcting may be regarded as subtypes of improving. So, if we call software PLCC activities to adapt, enhance, augment and correct errors 'maintenance,' the term 'maintenance' denotes quite different phenomena for hardware and software: improving a product for software while preserving and/or restoring the original state of a product for hardware. As the result, confusion is induced. The customary payment system for software 'maintenance' amplifies the confusion.

It looks almost mysterious that managers of acquiring organizations accept, although reluctantly, to bear the cost to fix errors for which their organizations are not responsible. One of seemingly reasonable explanations of this mysterious phenomenon is that error correction is typically classified into a single category named 'maintenance' along with adaption, enhancement, and augmentation and that managers are so accustomed with hardware maintenance for which they think it is natural and proper for them to pay.

Even for software, it looks natural and proper to managers that they bear the cost to adapt, enhance, or augment. Moreover, prestigious organizations such as ISO/IEC recommend acquirers to bears the cost of all kind of software maintenance after the warranty period, for example, under a blanket contract with fixed amount. So, they pay the bills for all kinds of software maintenance although they feel cheated. As the natural consequence, however, they frequently insist that all kind of PLCC endeavors are included in the contract. This is not the result that software developers or maintainers want. So, disputes arise. We cautiously propose not to use the term 'maintenance' at all for software.

A PLCC endeavor may have multiple purposes. Moreover, the purposes of a PLCC endeavor may change during the endeavor. So, it is not a good idea to classify PLCC endeavors according to their purposes. Instead, we propose to classify all PLCC endeavors rigorously into four types of modification, replacement, enlargement, and retirement. The proposed classification scheme is based on two dimensions reuse ratio and functional growth rate, and is exhaustive and mutually exclusive. The scheme is expected to help managers to anticipate the results of PLCC endeavors and the associated costs more easily and accurately. To let the expected benefit realized in the practice, however, it is necessary to review and improve the maintenance cost estimation models according to the classification scheme proposed in this paper.

Further researches on how to protect the acquirer's long-term interests are seriously needed. Researches on how information from individual PLCC endeavors should be fed back into and integrated with IT governance process are necessary too.

References

[1] Abrahamsson, P., Babar, M. A., and Kruchen, P., "Agility and Architecture: Can They Coexist?", *IEEE Software*, March/April 2010, pp. 16-22.

[2] Abran, A. and Nguyenkim, H., "Measurement

of the Maintenance Process from a Demand-Based Perspective", *Journal of Software Maintenance: Research and Practice*, Vol. 5, No. 2, 1993, pp. 63-90.

[3] Air Force Instruction 16-1001, *Verification, Validation and Accreditation*, June 1, 1996.

[4] Aldrich, J., Chambers, C., and Notkin, D., "ArchJava: Connecting Software Architecture to Implementation", *Proceedings of the 24th International Conference on Software Engineering*, *ACM*, 2002, pp. 187-197.

[5] ANSI/IEEE Std, *IEEE Standard Glossary for Software Engineering Terminology*, 1983, pp. 729-1983.

[6] Belady, L. and Lehman, M., "A Model of Large Program Development", *IBM Systems J.*, Vol. 15, No. 3, 1976, p. 225.

[7] Bellay, B. and Gall, H., "A Comparison of Four Reverse Engineering Tools", *Proceedings of the 4th Working Conference on Reverse Engineering, IEEE*, 1997, pp. 2-11.

[8] Boehm, B. W., *Software Engineering Economics, Prentice Hall PTR: Upper Saddle River*, NJ, USA, 1981.

[9] Chaptin, N., Hale, J., Kahn, K. R. and Jr., Tan, W. G., "Types of Software Evolution and Software Maintenance", *Journal of Software Maintenance and Evolution*, Vol. 13, No. 1, 2001, p. 3.

[10] Chen, L., Babar, M. A., and Nuseibeh, B., "Characterizing Architecturally Significant Requirements", *IEEE Software*, Vol. 40, 2013, pp. 38-45.

[11] Cleland-Huang, J., Hanmer, R. S., Supakhul, S., and Mirakhorli, M., "The Twin Peaks of Requirements and Architecture", *IEEE Software*, 2013, pp. 24-29.

[12] Dalgarno, M., "When Good Architecture Goes Bad", *Methods and Tools*, Vol. 17, 2009, pp. 27-34.

[13] Dekleva, S. M., "Software Maintenance: 1990 Status", *Journal of Software Maintenance: Research and Practice*, Vol. 4, No. 4, 1992, pp. 233-247.

[14] de Silva, L. and Balasubramaniam, D., "Controlling Software Architecture Erosion: A Survey", *Journal of Systems and Systems*, Vol. 85, 2012, pp. 132-152.

[15] Eick, S. G., Graves, T. L., Karr, A. F., Marron, J. S., and Mockus, A., "Does Code Decay? Assessing the Evidence from Change Management Data", *IEEE Transactions on Software Engineering*, Vol. 27, 2001, pp. 1-12.

[16] Fowler, M., *Refactoring: Improving the Design o Existing Code*, Addison Wesley, 1999.

[17] Gannod, G. C. and Cheng, B. H. C., "A Framework for Classifying and Comparing Software Reverse Engineering and Design Recovery Techniques", *Proceedings of the 6th Working Conference on Reverse Engineering, IEEE*, 1999, pp. 77-88.

[18] Glass, R. L., "Results of the First IS State-of-the-Practice Survey", *The Software Practitioner*, May 1996, pp. 3-4.

[19] Godfrey, M. W. and Tu, Q., "Growth, Evolution, and Structural Change in Open Source Software", *Proceedings of the International Workshop on Principles of Software Evolution*, 2001, pp. 103-106.

[20] Godfrey, M. W. and Tu, Q., "Evolution in Open Source Software: A Case Study",

Proceedings of the International Conference on Software Maintenance, IEEE Computer Society, Washington, DC, 2000, pp. 131-142.

[21] Grottke, M., Matias, R. and Trivedi, K. S., "The Fundamentals of Software Aging", *Proceedings of 1st International Workshop of Software Aging and Rejuvenation, IEEE*, 2008, pp. 1-6.

[22] Harris, D. R., Reubenstein, H. B., and Yeh, A. S., "Reverse Engineering to the Architectural Level", *Proceedings of the 17th International Conference on Software Engineering, ACM*, 1995, pp. 186-195.

[23] Hatton, L., "How Accurately Do Engineers Predict Software Maintenance Tasks?" *Computer*, February 2007, pp. 64-69.

[24] Helms, G. L. and Weiss, I. R., "Applications Software Maintenance: Can It Be Controlled?", *ACM SIGMIS Database*, Vol. 16, No. 2, 1984, pp. 16-18.

[25] Herraiz, I., Rodriguez, D., Robles, G., and Gonzalez-Barahona, J. M., "The Evolution of the Laws of Software Evolution: A Discussion Based on a Systematic Literature Review", *ACM Computing Surveys*, Vol. 46, No. 2, Article 28, 2013.

[26] Hochstein, S. and Lindvall, M., "Combating Architectural Degeneration: A Survey", *Information and Software Technology*, Vol. 47, 2005, pp. 643-646.

[27] Huang, Y., Kintala, C., Kolettis, N., and Fulton, N. D., "Software Rejuvenation: Analysis, Module and Applications", *Proceedings of International Symposium on Fault-Tolerant Computing, IEEE*, 1995, pp. 381-390.

[28] Hunt, B., Turner, B., and McRitchie, K., "Software maintenance Implications on Cost and Schedule", *Proceedings of Aerospace Conference, 2008 IEEE*, March 2008, pp. 1-8.

[29] IEEE, IEEE Std. 1219-1998, *IEEE Standard for Software Maintenance*, 1998.

[30] ISO/IEC, *ISO/IEC 14764(IEEE Std 14764-2006), Software Engineering-Software Life Cycle Processes-Maintenance* (2nd ed.), 2006-09-01.

[31] Israeli, A. and Feitelson, D. G., "The Linux Kernel as a Case Study in Software Evolution", *J. of System and Software*, Vol. 83, No. 3, 2010, pp. 485-501.

[32] Izurieta, C. and Bieman, J., "How Software Designs Decay: A Pilot Study of Pattern Evolution", *Proceedings of 1st International Symposium on Empirical Software Engineering and Measurement, IEEE*, 2007, pp. 449-451.

[33] Jacobson, I., *Object-Oriented Software Engineering*, Addison Wesley, 1992.

[34] Jones, C., *Software, Assessments, Benchmarks, and Best Practices*, Addison Wesley Longman, Inc., 2000.

[35] Jorgensen, M., Sjoberg, D. I. K., Kirkeboen, G., "The prediction Ability of Experienced Software Maintainers", in *Proceedings of 4th European Conference on Software Maintenance and Re-engineering*, 2000, pp. 93-99.

[36] Kemerer, C. F. and Slaughter, S. A., "Determinants of Software Maintenance Profiles: An Empirical Investigation", *Journal of Software Maintenance: Research and Practice*, Vol. 9, No. 4, 1997, pp. 235-251.

[37] Koch, S., "Evolution of Open Source Software Systems-A Large-Scale Investiga-

tion", *Proceedings of the International Conference on Open Source Systems*, 2005.

[38] Koch, S., "Software Evolution in Open Source Projects-A Large-Scale Investigation", *J. of Software Maintenance and Evolution Research Practice*, Vol. 19, No. 6, 2007, pp. 361-382.

[39] Koh, S., "Purposes and Types of Post Life Cycle Changes", *Proceedings of 13th International Conference on IT Applications and Management: Diversity and Collaboration in the Age of Digital Convergence*, Phitsanulok, Thailand, January 14-16, 2015, pp. 75-87.

[40] Koh, S., "Types of Post Life Cycle Changes and Evolution of Software", *Proceedings of 12th International Conference on IT Applications & Management: Culture and Humanities in the Digital Future*, Kenya, July 8-9, 2014, pp. 51-55.

[41] Koh, S., "A Software Architecture Life Cycle Model Based on the Program Management Perspective: The Expanded Spiral Model", *J. of Information Technology Applications and Management*, Vol. 20, No. 2, 2013, pp. 69-87.

[42] Koh, S., "An Extensive Model on Essential Elements of Software Architecture", *J. of Information Technology Applications and Management*, Vol. 19, No. 2, 2012, pp. 135-147.

[43] Koh, S. and Ji, K. S., "Contextual Models of Business Application Software Architecture", *J. of Information Technology Applications and Management*, Vol. 20, No. 3, Sept. 2013, pp. 1-18.

[44] Kooser, A. C., "Old Software Draining your IT Budget?", *Entrepreneur*, May 2005, p. 24.

[45] Lehman, M. M., "Laws of Software Evolution Revisited", *Proceedings of the European Workshop on Software Process Technology*, Springer-Verlag, London, 1996, pp. 108-124.

[46] Lehman, M. M., "Program, Life Cycles, and Laws of Software Evolution", *Proceedings of the IEEE* (Special Issue on Software Engineering) 1980, pp. 1060-1076; with more detail as "Programs, Programming and the Software Life-Cycle", in: *System Design, Infotech State of Art*, Rept. Se 6, No. 9, Pergamon Infotech Ltd. Maidenhead, 1981, pp. 263-291; reprinted as Chapter 19 in Lehman, M. M., L. A. Belady, Program Evolution-*Process of Software Change*, Academic Press, London, 1985.

[47] Lehman, M. M., "Laws of Program Evolution-Rules and Tools for Programming Management", Proceedings of the Infotech State of Art Conference, Why Software Projects Fail, 1978; also as Chapter 12 in M.M. Lehman, L. A. Belady, *Program Evolution-Process of Software Change*, Academic Press, London, 1985.

[48] Lehman, M. M., "Programs, Cities, Students, Limits to Growth?", in: *Imperial College of Science and Technology Inaugural Lecture Series*, Vol. 9, 1974, pp. 211-229; Also in M. M. Lehman, L. A. Belady, *Program Evolution-Process of Software Change*, Academic Press, London, 1985.

[49] Lehman, M. M., *The Programming Process*, IBM Res. Rept. RC2722, December 1969; also as Chapter 3 in M. M. Lehman, L. A. Belady, *Program Evolution-Process of Software*

Change, Academic Press, London, 1985.

[50] Lehman, M. M., and Belady, L. A., *Program Evolution-Process of Software Change*, Academic Press, London, 1985.

[51] Lehman, M. M. and Ramil, J. F., "Software Evolution-Background, Theory, Practice", *Information Processing Letters*, Vol. 88, 2003, pp. 33-44.

[52] Lehnert, S., Farooq, Q., and Riebisch, M., "A Taxonomy of Change Types and its Application in Software Evolution", *Proceedings of 2012 IEEE 19th International Conference and Workshop on Engineering of Computer-Based Systems*, 2012, pp. 98-107.

[53] Lewis, R. O., *Independent Verification and Validation: A Life Cycle Engineering Process for Quality Software*, New York: John Wiley and Sons, Inc., 1992.

[54] Lientz, B. and Swanson, B., *Software Maintenance Management*, Addison-Wesley, Reading, MA, 1980.

[55] Martin, R. J. and Osborne, W., "Guidance ofSoftware Maintenance", U.S. National Bureau of Standards, NBS Pub. 500-129, Dec, 1983.

[56] Mirakhorli, M., "Tracing Architecturally Significant Requirements: A Decision-Centric Approach", Proceedings of ICSE'11, May 21-28, 2011, Waikiki, Honolulu, HI, USA, pp. 1126-1127.

[57] Neamtiu, I., Xie, G., and Chen, J., "Towards a Better Understanding of Software Evolution: An Empirical Study on Open-Source Software", *J. of Software: Evolution and Process*, Vol. 25, No. 3, 2013, pp. 193-218.

[58] NF EN 13306, *Terminologies de la Maintenance*, June 2001.

[59] Niu, N., Su, L. D., Chen, J.-R. C., and Niu, Z., "Analysis of Architecturally Significant Requirements for Enterprise Systems", *IEEE Systems Journal*, 2013, pp. 1-8.

[60] OMG(Object Management Group), *Business Process Model and Notation* (BPMN), Ver. 2.0, OMG, 2011.

[61] Parnas, D. L., "Software Aging", *Proceedings of the 16th International Conference on Principles of Software Engineering*, Sorrento, Italy, 1994, pp. 279-287.

[62] Perry, D. E. and Wolf, A. L., "Foundations for the Study of Software Architecture", *ACM Software Engineering Notes*, Vol. 17, No. 4, 1992, pp. 40-52.

[63] PMI, *A Guide to the Project Management Body of Knowledge* (5th ed.), PMI, 2013.

[64] Ratkin, S. R., *Software Verification and Validation: A Practitioner's Guide*, Norwood, MA: Artech House, Inc., 1997.

[65] Riaz, M., Sulayman, M., and Naqvi, H., "Architectural Decay during Continuous Software Evolution and Impact of 'Design for Change' on Software Architecture", *Proceedings of the International Conference on Advanced Software Engineering and its Applications*, Springer, 2009, pp. 119-126.

[66] Robles, G., Amor, J. J., Gonzalez-Barahona, J. M., and Herraiz, I., "Evolution and Growth in large LIBRE Software Projects", *Proceedings of the International Workshop on Principles of Software Evolution*, 2005, pp. 165-174.

[67] Schulmeyer, G. G. and MacKenzie, G. R., *Verification and Validation of Modern Soft-*

ware-Intensive Systems, Upper Saddle River, NJ: Prentice Hall PTR, 2000.

[68] Sneed, H. M., "A Cost Model for Software Maintenance and Evolution", *Proceedings of the 20th IEEE International Conference on Software Maintenance* (ICSM'04), 2004.

[69] Stringfellow, C., Amory, C., Potnuri, D., Andrews, A., and Geor, M., "Comparison of Software Architecture Reverse Engineering Methods", *Information and Software Technology*, Vol. 48, 2006, pp. 484-497.

[70] vanGurp, J. and Bosch, J., "Design Erosion: Problems and Causes", *Journal of Systems and Software*, Vol. 61, 2002, pp. 105-119.

[71] Vasa, R., *Growth and Change Dynamics in Open Source Software Systems*, Ph. D. thesis, Swinburne University of Technology, Melbourne, Australia, 2010.

[72] Yip, S. W. L., Lam, T., Chan, S. K. M., "A Software Maintenance Survey", *Proceedings of the 1st Asia-Pacific Software Engineering Conference*, Tokyo, 1994, pp. 70-79.

Author Profile

Seokha Koh

Seokha Koh is the professor of the Department of MIS, Chungbuk National University. His current primary research areas include Software Quality Management, Business Process Modeling, Software Architecture, Project Management, and Software Engineering.

Man Pil Han

Man Pil Han is the doctoral-student of the Department of MIS, Chungbuk National University and CEO of Daesung Corp/D.S Package. He is interested in information systems and performance of small and medium business.