Information and Software Technology 52 (2010) 31–51

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Characterizing software architecture changes: A systematic review

Byron J. Williams [a], Jeffrey C. Carver [b,*]

[a] Department of Computer Science and Engineering, Mississippi State University, United States
[b] Department of Computer Science, University of Alabama, Box 870290, 101 Houser Hall, Tuscaloosa, AL 35487-0290, United States

ARTICLE INFO

Article history:
Received 10 December 2008
Received in revised form 9 July 2009
Accepted 13 July 2009
Available online 18 July 2009

Keywords:
Software architecture
Software maintenance
Change characterization
Software evolution
Systematic review
Software changes

ABSTRACT

With today's ever increasing demands on software, software developers must produce software that can be changed without the risk of degrading the software architecture. One way to address software changes is to characterize their causes and effects. A software change characterization mechanism allows developers to characterize the effects of a change using different criteria, e.g. the cause of the change, the type of change that needs to be made, and the part of the system where the change must take place. This information then can be used to illustrate the potential impact of the change. This paper presents a systematic literature review of software architecture change characteristics. The results of this systematic review were used to create the Software Architecture Change Characterization Scheme (SACCS). This report addresses key areas involved in making changes to software architecture. SACCS's purpose is to identify the characteristics of a software change that will have an impact on the high-level software architecture.

Contents

* Corresponding author. Tel.: +1 205 348 9829; fax: +1 250 348 0219.
E-mail addresses: bjw1@cse.msstate.edu (B.J. Williams), carver@cs.ua.edu (J.C. Carver).

0950-5849/$ - see front matter
doi:10.1016/j.infsof.2009.07.002

1. Introduction

Software change is inevitable. All software systems must evolve to meet the ever-expanding needs of its users. Therefore, it is vital for organizations to perform software maintenance in such a way as to reduce complications arising from changes and the potential for new bugs to be introduced by the change. Software developers need a comprehensive solution that helps them understand changes and their impact. This understanding is important because, as changes are made architectural complexity tends to increase, which will likely result in an increase in the number of bugs introduced [50,65,114]. *Architectural complexity* measures the extent to which the behavior of one component can affect the behavior of other components from an architectural standpoint [16]. A complex system is potentially less understandable for developers resulting in decreased quality and a system that is more difficult to maintain [13]. *Software quality* is the degree to which software possesses a desired combination of quality attributes [1]. Due to the number and frequency of changes that mature systems undergo, software maintenance has been regarded as the most expensive phase of the software lifecycle.

Late-lifecycle changes are changes that occur after at least one cycle of the development process has been completed and a working version of the system exists. These unavoidable changes pose an especially high risk for developers. Understanding late-lifecycle changes is important because of their high cost, both in money and effort, especially when they are due to requirements changes. Furthermore, these late-lifecycle changes tend to be the most crucial changes because they are the result of better understood customer and end-user needs. As these late changes are made, system complexity tends to increase. Different names have been given to this phenomenon of increasing complexity. Eick, et al., called the problem *code decay*. They examined a 15-year old system and found that it became much harder to change over time. One cause of this decay was the violation of the original architectural design of the system [50]. Lindvall, et al., called the problem *architectural degeneration*. This term is used to describe the deviation of the architecture They found that even for small systems the architecture must be restructured when the difficulty of making a change becomes disproportionately large relative to its size [93]. Parnas used the term *software aging* to identify the increased complexity and degraded structure of a system. He noted that a degraded structure or architecture would increase the number of bugs introduced during incremental changes [114]. And finally, Brooks stated that "all repairs tend to destroy the structure, to increase the entropy and disorder of the system…more and more time is spent on fixing flaws introduced by earlier fixes" [35].

Flexibility is a quality property of the system that defines the extent in which the system allows for unplanned modifications [116]. Flexibility is reduced when late changes draw the system away from its original design. There are many sources of late-lifecycle changes including: defect repair, adapting to changing market conditions or software environments, and evolving user requirements. Due to the time pressure resulting from these crucial late-lifecycle changes, developers often cannot fully evaluate the architectural impact of each change. As a result, the architecture degrades (i.e., becomes increasingly difficult to change), escalating the likelihood of faults and the difficulty of making future changes [50,84,114].

When dealing with late-lifecycle changes, it is important to focus on the *software architecture*, a high-level representation that defines the major structure and interactions of the internal components of a system and the interactions between the system and its environment [59]. When a change affects the architecture, the original architectural model must be updated to ensure that the system remains flexible and continues to function as originally designed. When an architectural change causes the interactions to become more complex, which in turn causes the system to be more difficult to change, the architecture is degenerating. *Architectural degeneration* is a mismatch between the actual functions of the system and its original design. Because architectural degeneration is confusing for developers, the system must undergo either a major reengineering effort or face early retirement [65].

To address these problems, developers need a way to better understand the effects of a change prior to making it [100]. The high-level goal of this research is to:

> *Identify and characterize the types of changes that affect software and develop a framework for analysis and understanding of change requests.*

This paper presents a systematic literature review of software change. The goal of the review was to identify and characterize software architecture changes to determine the types of changes that impact software architectures. A systematic literature review is a formalized, repeatable process in which researchers systematically search a body of literature to document the state of knowledge on a particular subject. A systematic review provides the researchers with more confidence in their conclusions compared with an ad hoc review. A needs assessment conducted prior to the review indicated several key areas that must be addressed to improve the software change process.

- *Change understanding and architecture analysis*: Prior to making a change, it is important for a software developer to understand how it will impact the architecture. A change analysis tool should allow the developer to analyze a change prior to implementation to understand the change, the architecture, and how the change fits with the architecture [11,25,53,113].

- *Build historical baseline of software change data: The ability to compare a* change request to a system's history will provide insight into the change impact, difficulty, and required effort. Recording information about the change type, its impact on the architecture and the effort required will provide insight into future changes [85].
- *Group changes based on impact/difficulty:* Change requests can be grouped based on their characteristics. Similar changes should exhibit a similar impact on the system. Heuristics can be developed to handle certain types of changes [71,112].
- *Facilitate discussion among developers*: methods that facilitate discussion among the development team are useful in achieving consensus on the implementation approach. A characterization scheme should facilitate consensus building by providing a list of the items to discuss to prevent the change from violating the planned architectural structure [64,100].
- *Facilitate change difficulty/complexity estimation*: The characterization scheme should allow a developer to determine change complexity as a function of type and size. Characterizing the context of the change request (i.e., a description of influencing factors external to the request) should also help facilitate difficulty estimation because certain types of changes may be more difficult in certain domains than others [13,77].

The output of the review provides insight into the architectural change process and describes the effects that changes can have on architecture. As a result of the review findings, the Software Architecture Change Characterization Scheme (SACCS) was created. The attributes of the proposed scheme were extracted from change taxonomies and associated change characteristics identified during the review.

The remainder of this paper is organized as follows. Section 3 discusses the review method. The results of the systematic review are summarized in Section 4. Section 5 reports on the development of SACCS. Section 2 describes background and related work. Finally, the results are discussed and future work is presented in Section 6.

2. Background

This section discusses previous work about software change, late changes and change classifications. Then it provides the motivation for conducting this systematic review.

2.1. Software change

Software change is a long-studied topic. Manny Lehman, a pioneer of the study of software changes, developed the Laws of Software Evolution [84]. These laws describe recurring issues related to the evolution of E-type software. An E-Type system is one that functions in the real world and therefore must continually evolve to maintain user satisfaction [87]. For our current work, laws I, II, VI, and VII are the most relevant and are described in more detail in this section.

Law I, *Continuing Change*, states that software undergoes never-ending maintenance and development that is driven by the mismatch between current capability and environmental requirements [84]. This mismatch could be the result of changes in protocols and standards used to communicate with other systems. It could also be the result of changes in hardware or the need to more efficiently utilize hardware resources. An understanding of the reasons for a change supports the development of systematic processes to handle change.

As systems change, they will become more complex if those changes are not properly handled. Law II, Increasing Complexity, captures this situation. This law simply states that changes imposed by system adaptation lead to an increase in the interactions and dependencies among system elements. These interactions may be unstructured and increase the system entropy. If entropy is not properly handled, the system will become too complex to adequately maintain. Law II is one of the primary reasons why the maintenance phase is typically the most expensive phase of software development. In order to better manage system complexity, developers need improved ways of understanding changes and how to incorporate change into system architectures.

It has been shown that the number of system modules increases with each incremental system release [84]. Law VI, *Continuing Growth*, focuses on user needs by stating that the functionality of software systems must continually increase to maintain user satisfaction over the lifetime of a system [83]. This law, while similar to Law I, reflects a different phenomena. The law addresses the tendency of the user base to become increasingly sophisticated and demand a more robust set of features resulting in software growth to meet their needs. This growth includes adaptation of features that do not adequately meet user needs.

The previous three laws induce Law VII, *Declining Quality* [83]. As systems continually change (Law I), complexity increases (Law II). The introduction of new features to a system causes it to grow (Law VI). These factors all reduce the perceived quality of a system. When the quality of the system is reduced, it becomes more expensive to maintain because of an increase in the number of problems encountered by users. These changes are likely to further increase the complexity and growth of the system which will, in turn, further reduce the quality [83]. This cycle results in a continuous downward spiral of quality.

The Laws of Software Evolution have been extensively studied [17,18,37,38,47,57,63,81,86,88,107,115,136]. In understanding these laws, and the necessity of software change, researchers have developed methods for handling changes, e.g., using change classification schemes, performing impact analysis, and building effort prediction models. These methods are continuing to improve. As more research is performed to understand changes, more can be done to help engineers implement changes. Practitioners then will not have to suffer from an uncontrollable increase in complexity or decline in quality [30].

2.2. Change classification

Change classification schemes have been used to qualitatively assess the impact and risks associated with making certain types of changes [31,32]. Software change classification schemes also allow engineers to group changes based on different criteria, e.g. the cause of the change, the type of change, the location of the change, the size of the code modification or the potential impact of the change. Another benefit of change classification is that it allows engineers to develop a common approach to address similar changes thereby reducing overall effort compared with addressing each change individually [111].

Lientz and Swanson's work identified the frequency of the different types of maintenance activities performed by software development organizations [90]. Based on their work and work by Sommerville, four major types of changes were identified. *Perfective* changes result from new or changed requirements. These changes improve the system to better meet user needs. *Corrective* changes occur in response to defects. *Adaptive* changes occur when moving to a new environment or platform or to accommodate new standards [128]. Finally, *preventative* changes ease future maintenance by restructuring or reengineering the system [106].

The architectural change process identified by Nedstam describes the change process as a series of steps [108]:

1. Identify an emergent need.
2. Prepare resources to analyze and implement change.
3. Make a go/no-go feasibility decision.
4. Develop a strategy to handle the change.
5. Decide what implementation proposal to use.
6. Implement the change.

An architectural change characterization scheme will address steps 2, 3, and 4 by helping developers conceptualize the impact of a proposed change by characterizing the features of the change request.

3. Research method

A systematic review is a means of identifying, evaluating and interpreting the available research related to a research question, topic area, or phenomenon. The main purpose for conducting a systematic review is to gather evidence on which to base conclusions. They are commonly used to support or contradict claims made by researchers, identify gaps in existing research, provide motivation for new research, and supply a context for the new research. A systematic review consists of planning, conducting and reporting the review [72]. Within those three phases are the following steps:

1. Identification of the need for a systematic review.
2. Formulation of a focused review question.
3. A comprehensive, exhaustive search for primary studies.
4. Quality assessment of included studies.
5. Identification of the data needed to answer the research question.
6. Data extraction.
7. Summary and synthesis of study results (meta-analysis).
8. Interpretation of the results to determine their applicability.
9. Report-writing.

We used the systematic review protocol template prescribed by Biolchini to perform the review [24]. This review protocol was modified to encompass the requirements of this review and is described in the next section. The remainder of this section describes the steps performed to complete the review.

3.1. Research questions

The review goal was to identify software change characteristics that affect architecture. We began by defining several research questions to focus the review. The high-level question was:

> *Can a broad set of characteristics that encompass changes to software architectures be identified using the current software engineering body of knowledge and be used to create a comprehensive change assessment framework?*

This research question was then refined to 5 more specific questions. These questions, along with the motivation for each one, are shown in Table 1.

3.2. Sources selection and search

The primary studies used in this review were obtained from searching databases of peer-reviewed software engineering research that met the following criteria:

- Contains peer-reviewed software engineering journals articles, conference proceedings, and book chapters.
- Contains multiple journals and conference proceedings, which include volumes that range from 1970 to 2007.
- Used in other software engineering systematic reviews [29,61,68,73,99].

The resulting list of databases was:

- ACM Digital Library
- Google Scholar
- IEEE Electronic Library
- Inspec
- Scirus (Elsevier)
- SpringerLink

The database searches resulted in a large number of candidate papers. The inclusion/exclusion criteria shown in Table 2 were used to narrow the search to relevant papers. Papers that addressed specialized areas of software development or any non-object-oriented software were also excluded because the goal was to generalize these results to a broad range of domains. The *population* of this study is the domain of software maintenance. *Intervention* includes applying changes throughout the maintenance process. The *outcomes* of the search represents different types of

Table 1
Research questions.

Research question	Motivation
1. What are the attributes of existing software change classification taxonomies?	This question provides a starting point for creating a framework for change assessment. The answers to this question present the basis on which to define, build, and refine the attributes of the scheme
2. How are software architecture elements and relationships used when determining the effects of a software change?	One requirement of developing the framework is to understand the role of architecture in a developer's assessment of change impact. The answer is important in understanding architectural characteristics that affect change implementation difficulty
3. How is the architecture affected by functional and non-functional changes to the system requirements?	Software architectures are important in exhibiting the non-functional requirements of a system. The impact of an architectural change due to a functional requirement may be less important. The goal here was to differentiate, if possible, the architectural effects of changes to functional and non-functional requirements
4. How is the impact of architecture changes qualitatively assessed?	Developers often have differing views on the best way a change should be implemented to a system. The internal processes that developer's use when assessing a change is an important abstraction to understand in developing a change assessment scheme
5. What types of architecture changes can be made to common architectural views?	Understanding how an architecture changes is important if your goal is to provide developers with a list of alternatives for making decisions about changing the architecture. Having this list of the possible changes can lessen the cognitive load on the developer to a set of choices given the context of the request

Table 2
Inclusion/exclusion criteria.

Inclusion Criterion	Exclusion Criterion
• Papers that address change classification at any level of abstraction (i.e., class, package, subsystem, and architecture) • Papers that identify procedures and techniques for change impact analysis • Papers that discuss the effects of changing software architectures • Empirical studies of software changes • Case studies of software change frameworks and change assessment methodologies • Experience reports detailing software changes	• Papers that are based only on expert opinion • Short papers, introductions to special issues, tutorials, and mini-tracks • Studies presented in languages other than English • Studies whose findings are unclear and ambiguous • Papers that describe changes to aspect-oriented software, self-adaptive software systems, embedded systems, and dynamic software architectures

characteristics identified when making a change to a software system. The *experimental designs* of the studies include empirical studies, case studies, and experience reports.

This inclusion/exclusion criteria was applied by:

1. Reading the title to eliminate any irrelevant papers.
2. Reading the abstract and keywords to eliminate additional papers whose title may have fit, but abstract did not relate to any of the research question.
3. Reading the remaining papers and including only those that addressed the research questions.

The research questions were reduced to a series of search strings that were executed in the selected databases. The search strings can be found in Appendix A, Table A.1. These searches returned thousands of papers. Google Scholar returned greater than 1 million results and the ACM Digital Library returned as many as 30,000 for one search. Each database returned the most relevant searches first. We used this factor to eliminate results that were not relevant. We scanned each title in order until papers were reached that did not have any relevance to this study. This includes papers that were from different technical domains and papers where software engineering was not the focus. Using this approach, the initial results were filtered down to 2752 papers. This number was then further reduced to 523 after detailed analysis of the titles, and then to 220 upon reading the abstract. These 220 papers were read and 130 were chosen based on the exclusion/inclusion criteria. Of the 130 primary studies, 36 were published in scholarly journals and 94 in conference proceedings. In addition to the primary studies, 8 books were referenced to provide additional background data on software architectures and software [12,27,45,55,84,90,127,128]. Four technical reports were also found that met the inclusion/exclusion criteria [16,56,103,129] and three standards documents [2–4]. Prior to conducting the systematic search, we were aware of a number of papers that were relevant. As an indicator of the completeness of the review, we again found all of those papers through our search. [7,20,22,30,41,44,51,65,66,76,77,82–84,94,96,104,106,108,120,122,130,138]. Table 3 lists the paper distribution by source.

3.3. Data extraction and quality assessment

A data extraction form was used to extract relevant data from each paper. The form includes the superset of all data items examined for each study. Every paper did not provide information for each data item, but if the information was included, it was recorded in the form. The data extraction form is show in Table 4.

Using the data extraction form, one of us reviewed all papers and extracted data. Then, the other one independently reviewed and extracted data from a sample of the papers. We then analyzed our extracted data for consistency. We found that we had consistently extracted information from the sample of papers given that we both knew the aims of the research and criteria for paper selection. When there were, at times, minor discrepancies in the information extracted, we reviewed the extraction forms and discussed the findings to resolve any inconsistencies or differing interpretations in the extraction process. As a result, there were no disagreements after these short discussions. This process is consistent with the process followed in previous systematic reviews [61,68,72,99].

The papers that were included in the results were checked using the quality criteria below.

- The goals of the research are clearly defined.
- The studies methods were well defined and deliberate.
- The study environment and contexts are clearly stated.
- The observations/results support the conclusions.
- The study addresses threats and minimizes bias.
- The study can be repeated.
- The research add value to the software engineering community.

There were two studies removed due the quality criteria [59,89]. The data extracted from all papers was then synthesized to answer each question.

4. Reporting the review

This section details the findings of the review. The results of the review are presented to answer each Research Question listed in Table 1. This information provides the basis for the creation of the Software Architecture Change Characterization Scheme described in Section 5.

4.1. Research question 1: what are the attributes in existing software change classification taxonomies?

Of the studies found that reported on change classification, several focused on classifying source code changes, others identified organizational, management, and external factors as influencers of change implementation. There were also studies that examined the features of the change requests to determine their effects on the system. And finally, there were change taxonomies that focused on how system designs would be affected by a change. Based on an examination of the general characteristics of change classification taxonomies, the major categories of change taxonomies are listed below. For each category, we provide the related findings based on the high-level change classification types that we identified.

4.1.1. Prescriptive change types

Prescriptive changes 'prescribe' a course of action that is typical for addressing that change type. Change taxonomies of this category generally originated from the work done by Lientz and Swanson [90,133]. Much of their work was incorporated in the seminal

Table 3
Paper distribution.

Source	Count	%
International Conference on Software Maintenance	21	16.15
IEEE Transactions on Software Engineering	10	7.69
International Conference on Software Engineering	9	6.92
European Conference on Software Maintenance and Reengineering	7	5.38
IEEE Symposium on Software Metrics	6	4.62
International Symposium on Principles of Software Evolution	6	4.62
Journal of Software Maintenance and Evolution: Research and Practice	6	4.62
Working IEEE/IFIP Conference on Software Architecture	4	3.08
IEEE Software	3	2.31
Information and Software Technology	3	2.31
International Conference on Computer Systems and Applications	3	2.31
International Workshop on Mining Software Repositories	3	2.31
Journal of Systems and Software	3	2.31
Annual NASA Goddard/IEEE Software Engineering Workshop	2	1.54
Asia Pacific Conference on Software Engineering	2	1.54
Communications of the ACM	2	1.54
Empirical Software Engineering	2	1.54
IEEE International Workshop on Software Evolvability	2	1.54
International Conference on Automated Software Engineering	2	1.54
International Conference on Quality Software	2	1.54
International Conference on Software Engineering and Knowledge Engineering	2	1.54
International Software Architecture Workshop	2	1.54
International Symposium on Empirical Software Engineering	2	1.54
Working Conference on Reverse Engineering	2	1.54
ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications	1	0.77
Annual Hawaii International Conference on System Sciences	1	0.77
Australian Software Engineering Conference	1	0.77
Bell Labs Technical Journal	1	0.77
Conference of the Centre for Advanced Studies on Collaborative Research	1	0.77
Cutter IT Journal	1	0.77
EUROMICRO Conference on Software Engineering and Advanced Applications	1	0.77
IEEE International Requirements Engineering Conference	1	0.77
IEEE International Symposium on Requirements Engineering	1	0.77
IEEE International Symposium on Visual Languages	1	0.77
IEEE Region 10 International Conference	1	0.77
IEEE Symposium and Workshop on Engineering of Computer-Based Systems	1	0.77
International Conference on Information Systems	1	0.77
International Software Process Workshop	1	0.77
International Conference on Applying the Software Processes	1	0.77
International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing	1	0.77
International Journal of Software Engineering and Knowledge Engineering	1	0.77
International Process Support of Software Product Lines Software Process Workshop	1	0.77
International Workshop on Program Comprehension	1	0.77
Proceedings of the IEEE	1	0.77
Science of Computer Programming	1	0.77
Software – Practice and Experience	1	0.77
Software Process Improvement and Practice	1	0.77
Workshop on Unanticipated Software Evolution	1	0.77
Total	130	100%

Table 4
Data extraction form.

Data item	Description
Focus of article	State the main objective of the article
Attributes of change classification scheme and definitions	List any categories of changes identified by the article and their definitions
Description of changes	Describe the different types of changes that were made
Impact analysis techniques	Describe how the techniques are used to perform impact analysis
Impact quantification	Describe how the impact was measured
Software architecture impact	Describe how the system architecture was impacted by any change
Changes to software architecture	Describe what exactly changed in the system architecture
Quantitative Results	Record quantitative empirical results
Qualitative Results	Record qualitative empirical results
Related References	Record additional references that pertain to the research questions

text on software engineering by Sommerville [128]. They identified four types of maintenance activities: *perfective*, *corrective*, *preventative*, and *adaptive*. There have been numerous studies on these four types (or a subset of the four) that attempt to measure frequency and potential impact [6,19,30,50,66,69,103,105,106,119,126].

Other researchers have expanded on the four major change types. *Evaluative*, *consultive*, *training*, *updative*, *reformative*, *performance*, *groomative*, *reductive*, were identified in addition to the general change types called *enhancive*, *adaptive*, and *corrective* [40]. *Retrenchment*, *retrieving*, *prettyprinting*, and *documentation* were later added to the adaptive and corrective change types [91]. Some have analyzed changes to design patterns and created design evolution patterns: *intensive evolution* (e.g., requirements change, bug fix and design improvement), *extensive evolution* (e.g., new requirements and accommodating new operating environment) and *evolution operations* (e.g., module replacement and connection change) [8].

4.1.2. Source code changes

Early research focused specifically on source code changes. One study evaluated *signature* changes, that is, small changes to function names, parameters, or orderings in source code [70]. Another source code taxonomy that examined the results of *adding*, *deleting*, and *modifying* fields, methods, and classes [122]. Atomic changes and their affect on code structures such as *scope changes*, *inheritance deviation*, *signature changes*, *modifier*, *attribute*, *class declaration*, *interface* and *variable changes* have also been analyzed in detail [42,43,54,76]. Source changed that were frequently applied were observed and classified based on their causes including *introduction of duplicated code*, *repositioning a code fragment*, and *temporarily adding a code fragment* [139].

4.1.3. Organizational influence

External factors have been identified in change taxonomies as a way of determining how the development organization influences change management and implementation. For example, developer experience is a characteristic of the organization responsible for implementing a change request [49,92]. The Prism Model of changes classified changes based on their effect on environmental infrastructures (change and dependency structures). The dependency structures outlined in the Prism Model defined representation of factors, which included *people*, *policies*, *laws*, *processes*, and *resources* that affect change implementation from the organizational standpoint. This change structure facilitated the classification, recording, and analysis of change data [94,95].

Changes have also been characterized based on their *origin*, *cause*, *process elements*, *phase*, *kind of change*, and *modifier* (developer responsible) [109]. Others have found organizational profiles that are important in understanding change including *project manager, participants, contractual-constraints, project size, external suppliers, customers, operating-platform,* and *implementation-languages* [46]. Finally, Lam identifies ten change management issues that are essential to an effective change management process including the *importance of stakeholder consensus* in making changes and assessing risk [78].

In summary, each of the papers discussed in this section provides varying views of characterizing software changes. This diverse compilation of attributes are a means of classifying the features of a change request and interpreting its effect on the system.

4.2. Research question 2: how are software architecture elements and relationships used when determining the effects of a software change?

Software architectures are described in terms of components and connectors, modules and relationships, and the topology that manages the architectural entities. Software architectures exhibit certain properties such as coupling and cohesion. Coupling can be used to analyze the complexity of the architecture. Changes to architectures can positively or negatively affect *coupling.* Therefore, the change can affect system *complexity* or *understandability* [33,34,121]. Architectural degeneration is a phenomenon that occurs as changes increase coupling, thereby increasing complexity [65]. Architectural evaluations have been used to determine the relationship of changes to: coupling between modules and coupling between module classes [93,137,138]. In these instances, coupling affected complexity and understandability. A distinction can also be made between *design space* changes (logical structure, interface, and package) and *implementation space* changes (build components and header files) [143]. Finally, modules can be characterized by the frequency of change. Core architecture modules infrequently change while non-core or new modules are changed more frequently [60].

Another area of focus is characterizing evolution. The "Phasic Analysis" technique identified six evolution profiles describing how the architecture changes over time: *intense evolution*, *rapidly developing*, *restructuring*, *slowly developing*, *steady-state*, and *pending*. [141]. Another framework, created to facilitate component replacement in long-lived architecture, helps to determine whether a component should be *replaced* or *adapted to new technology*. The framework determines the quality properties defined by the architecture that will be affected when making a change such as performance issues, stability, scalability, and compatibility [117]. This research shows how different types of architecture changes can impact the maintenance process. A framework that transformed architecture patterns using transformation rules and exhibited transformational behaviors was developed and includes the following attributes: *superposition*, *conditional superposition*, *substitution*, and *conditional substitution* [15]. Others have pointed out the importance of assessing the impact of *dynamic architecture properties* in addition to *static architecture properties* [52].

All of the topics discussed in this section used architecture characteristics to assess and make determinations about maintenance. Several of the papers identified express the importance of the logical decomposition at the architectural level and others address runtime characteristics. These two facets of software architecture are important when determining how a system may evolve.

4.3. Research question 3: How is the architecture affected by functional and non-functional changes to the system requirements?

It has been shown that for many systems, that the majority of changes are requested in order to improve quality and enhance functionality [106]. Software architectures have been used to document how quality attributes and non-functional requirements are fulfilled [39,131,132]. When a modification affects a non-functional attribute (e.g., increasing system performance), the architecture is often affected also [23]. The effect of functional changes is not as obvious. Research Question 3 was asked to determine the effect that changing a system's architecture has on system quality and functional enhancements.

A software change can be a *strictly functional* change (affecting only user-observable attributes), a *strictly architectural* change (affecting only the architecture, unnoticeable to the user), or an *architectural/functional* change (a mix of the two) [108]. The three major types of evolution, considering source code features, that have the greatest architectural effect are: *interface evolution*, *implementation evolution*, and *structural evolution* [134]. These categories correspond to the strictly functional (interface evolution), architectural/functional (implementation evolution), and architectural (structural evolution) changes. Although strictly functional changes do not impact the architecture, the architecture does determine the location of the change. Therefore, architectural assessment is important for all three types of changes.

Functional changes may not result in a structural change to the architecture, but they do affect the portion of the architecture that is responsible for providing the specified feature. Therefore, most architecture modules can be characterized based on the features they provide. There are six functional software areas: *data handling* (data formats, record segments, databases or files, and establishment of parameters), *control flow* (references to changes in logic and program structure), *initialization* (source code modifications establishing constraints or initial data values), *user interface* (modifications of human–computer interfaces), *computation* (modifications for equations and functions), and *module interface* (changes in communication links between modules and/or submodules) [18,124]. Another way to look at functional changes is to determine the impact of the change on a module. The important impact factors include: *configuration file changes*, *data changes*, *functionality changes to the source code*, and *architecture changes* which were additions and/or deletions to architecture modules and connections [98].

Strictly architectural changes are those that have an impact only on the system architecture. These changes include *refactoring* and *restructuring* the architecture to enhance quality attributes [55,101]. They refer to *internal changes* that do not modify the external functionality of the system [55,67]. A preventative change is also a change that is strictly architectural in terms of its impact. These changes involve modifying an architecture component to 'prevent' problems in the future. These changes improve on non-functional quality attributes such as understandability, modifiability, and complexity [106].

In summary, software architectures have a substantial role in changes that address non-functional requirements (i.e., quality attributes). They play a lesser role in addressing purely functional changes. There exists a spectrum of changes ranging from purely architectural changes, to architectural/functional changes, and purely functional changes. Purely architectural changes consist of refactorings and changes to system structure. Purely functional changes affect some user-observed feature.

4.4. Research Question 4: How is the impact of architecture changes qualitatively assessed?

Change impact analysis is important because it can also assist in determining the amount of effort required to implement the change [111]. Thus, impact analysis can be approached in two ways. The most common way is to directly determine which code should be modified. The other way is to determine the consequences of the modification(s) [11]. Change impact can be *direct*, modules are affected because of a specific relationship to a module that will change, or *indirect*, a module is affected because of dependencies on the module that changes [26].

In terms of software architectures, determining the impact of a change can be challenging. Architecture modules can contain other modules, packages, and classes. Assessing change impact when architectures are affected involves highly complex structures and external factors not present when assessing impact from a strictly source code level. A developer must determine the underlying mechanisms of change and answer questions such as where (location of change), when (temporal properties), what (system properties), and how (change support) at the system and organization level [36]. Developers must also assess the abstraction level where the change takes place (i.e., module, subsystem, design unit, architecture, and systems of systems) [102].

Architecture impact analysis can be performed by scaling up low-level, source analyses [134]. Program traces and module dependency techniques are performed to determine which modules must be changed along with the target module. Impact analysis can also be performed dynamically by focusing on the module that executes after the most recently changed module [10]. Architectural impact analysis can also be performed statistically by determining the probabilities that a module change is required given that another module has changed [5].

In assessing architecture impact, subjective ratings based on developer experience are often used. For example, change impact can be *cosmetic* (trivial), *local* or *global* (significant impact) [97]. The influence of a requirement change can be *weak*, *average*, or *strong* [112]. Finally, the impact of a change to various classes can be characterized as *low*, *medium*, *high* or *no impact* [111].

Lassing et al. created a subjective impact scale in their study of architecture flexibility. The scale included four levels:

1. No impact.
2. Affects one component.
3. Affects several components.
4. Affects the software architecture.

They also pointed out the significance of analyzing *changes to the micro architecture* (internal components) and *changes to the macro-architecture* (external components) [79,80].

In summary, the results for Question 4 show that change impact analysis at the architecture/design level is subjective and involves determining which system modules will be affected when making a change. The subjectiveness of the approach allows developers to assess the level of impact using ordinal rating systems that require extensive developer experience to provide accurate results.

4.5. Research Question 5: What types of architecture changes can be made to common architectural views?

The architecture views used to describe software provide the architect with a means of explaining the architecture to stakeholders. Each view presents different aspects of the system that fulfill functional and non-functional requirements. There are many different architectural styles. At a very high-level, architectures are described in terms of their *logical (static) structure* and their *runtime (dynamic) structure* [58]. The logical views include; *dependency relationships, layers, inheritance structure, module decomposition,* and *source structure abstractions*. The runtime views include: *control flow processing, repository access, concurrent processes, component interaction, distributed components,* and *component deployment abstractions* [14,45,127].

There are many ways that architecture views can be modified. Logical changes affect system structure and consist of changes to *systems*, *subsystems*, *modules*, *packages*, *classes*, and *relationships*. Class hierarchy changes consist of modifications to inheritance views. *Class signature changes* describe alterations to system interfaces [48]. Change can be made to UML diagrams where each diagram type will signify the nature of changes made to it: *class diagrams* (i.e., add/delete attributes, change attribute, add/delete method, change method, add/delete relationship, change relationship, add/delete class, and change class), *sequence diagrams* and *state charts* [31]. A more general description includes changes to *entities* (i.e., classes and modules), *relations* and *attributes* [141,142]. Other types of architecture changes include: *kidnapping*, *splitting*, and *relocating*. Kidnapping is moving an entire module from one subsystem to another. Splitting involves dividing the functions of a module to two distinct modules. Relocating involves moving functionality from one module to another [135].

Runtime changes are identified based on changes to processing entities, the connections between the processes, and connections between remote components. Changes to components can have *causal dependencies* (i.e., behavior in one component causes a

Table 5
SACCS exclusion criteria.

Steps	Title	Definition
1	Subset of another attribute	The attribute is a more specific instance of an attribute that is include in SACCS
2	Not directly relevant to focus on software architecture	The attribute does not directly impact the process of changing the architecture
3	Not relevant to scope and goals of study	The attribute focused on characteristics that are outside the scope of this study
4	No extensive literature backing	The attribute did not have support from more than three sources
5	Proposed outcome of study	The attribute is an output of the characterization rather than an input.

Table 6
Overall impact scale.

Rating	Name	Description
0	No impact	The property will not be affected by the change request
1	Cosmetic impact	The property will be minimally effected with only a surface level impact
2	Minor impact	The property must be considered when planning the implementation of the change request
3	Substantial impact	This property will require considerable attention during the planning, implementation, and validation of the change request
4	Major focus of change	This property is one of the primary reasons for the change request in will require extensive

behavior in another component) and *ordering dependencies* (i.e., where a specific ordering relation has to be maintained between two or more component behaviors). These changes included *adding/deleting components*, *adding new components that refine existing components*, and *adding/deleting connections and component bindings* [28]. It is important to understand the evolution of architecture components and the communications between them [62]. There are taxonomies that also describe component changes. These changes included *adding*, *deleting*, *modifying*, or *substituting* components, connectors, ports, and services [9,118,123,125].

4.6. SACCS exclusion criteria

In the previous sections, we identified a large number of potential attributes for the SACCS. While each attribute has some value, including them all in the SACCS was not feasible due to the number of attributes that were available. Therefore, we created the exclusion criteria to systematically decide whether each attribute should be included. We used the five steps described in Table 5 as a method of filtering out those attributes that should be excluded from the SACCS.

In summary, there are many ways to reflect a change to the architecture in the architecture diagrams and in the source structure. The literature describes many ways to change high-level architecture components and their interactions. Each element in an architecture diagram has the potential to be affected by a software change. The next section presents the attributes that were used to create the characterization scheme.

5. Software architecture change characterization scheme (SACCS)

After reducing the overall set of identified attributes using the exclusion criteria defined in Table 5, we organized those attributes into a characterization scheme. The goal of this scheme was to assist developers in making decisions about how to address a change request.

5.1. Characterization scheme overview

SACCS was designed to capture the effects of changes to architecture and provide a structured approach for impact analysis. To use SACCS, a developer characterizes a change request beginning with high-level characteristics then progressing to more detailed characteristics. The high-level characteristics describe the change's motivation, type, size, impact on static, impact on dynamic properties and effect on requirements (functional and non-functional). The detailed change characteristics identify specific changes that must be made to the major architectural views. The two-level hierarchy for SACCS addresses both the context of the change request and its impact on the software system.

SACCS is organized as a set of characteristics that group together two or more related attributes. All of the characteristics and attributes originated in the literature discussed in Section 4. The remainder of this section discusses the change characteristics and associated attributes in detail. For each characteristic identified, the attributes are shown in rectangles in the figure on the left and listed in the table on the right along with the reference from which they were drawn. Any attribute that is a subset of an included attribute is listed along with the figure. Finally, for completeness, any attribute that was not included in SACCS is listed in Table 19 along with an explanation of its exclusion.

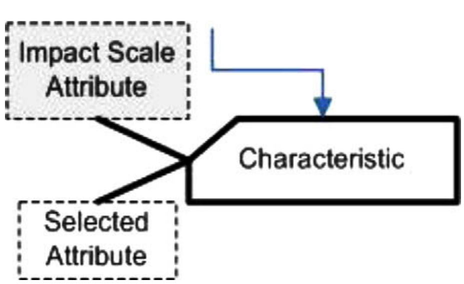


Fig. 1. General Characterization Shapes Key

5.2. General characteristics

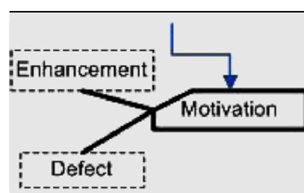
These high-level characteristics are used to describe how a change affects the system and development environment. In the following subsections, each characteristic is described in detail. In the description, the characteristic is listed in italics and the attributes are in bold. In the figure that follows each characteristic, the shape with the bold outline is the general characteristic. The shapes outlined with a dashed line are the values (attributes) that can be selected for each characteristic. Attributes that are shaded use the Overall Impact Scale (Table 6). Fig. 1 provides an example of how shapes are used in the scheme.

The Overall Impact Scale (Table 6) is used to determine the extent of the effect of each shaded attribute include in SACCS. This scale ranges from a rating of '0' (no impact), meaning the change will not have an effect on that attribute to '4' (major focus of change), meaning that the change will drastically affect that attribute [46,80,97,111,112].

5.2.1. Motivation

The first characteristic is the *motivation* for the change (Table 7). The change can be motivated by either the need for an *enhancement* (i.e., to improve the system) or in response to a *defect* (i.e., resulting from an error, fault, or failure) [128]. The relative frequency of defects vs. enhancements will, over time, provide insight

Table 7
Motivation characteristic.



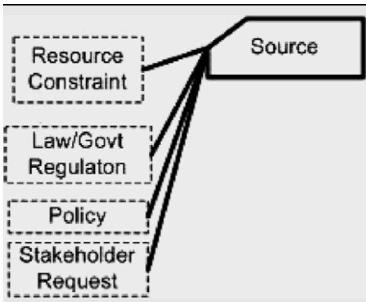
Enhancement [74,128]
Defect [74,128]

into system maintenance. An increase in the relative number of defects over time may suggest that the system quality is declining because more defects are introduced during the maintenance process [21].

5.2.2. Source

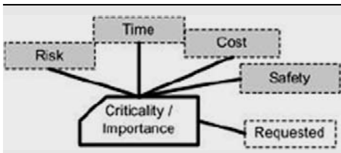
The *source* characteristic classifies the origin of the change request as one of four types (Table 8). First, the *resource constraint* attribute indicates that the source is a change in available resources or development environment (e.g., reduction in memory available, reduction in available communications protocols, or reduction in budget). Second, a change in a law or government regulation that affects the software's domain is classified as *law/government regulation*. Third, the source could be a change in organizational *policy*. Finally, the source could be a *stakeholder request* to address changing needs.

Table 8
Source characteristic.



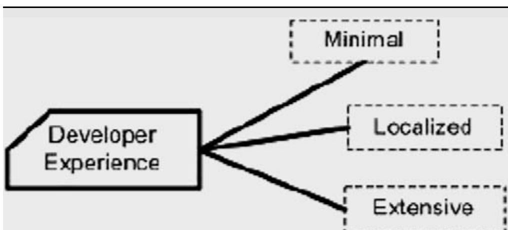
Resource constraint [94,95]
Law/government regulation [94,95]
Policy [94,95]
Stakeholder request [94,95,109]

Table 9
Criticality/Importance Characteristic.



Risk [78]
Time [110]
Cost [46,110]
Safety [46]
Requested [94,95,109]

Table 10
Developer experience characteristic.



Minimal [49,92,94,95,110]
Localized [49,92,94,95,110]
Extensive [49,92,94,95,110]

5.2.3. Criticality/importance

The *criticality/importance* characteristic contains five attributes that dictate the consequence of making the change (Table 9). *Risk* indicates that a change that has a greater than normal risk of failure or poses external risk to the organization. *Time* indicates that the change must be implemented within a shorter than normal timeframe. The *cost* attribute indicates that the change has greater than normal budget/resource constraints. The *safety* attribute indicates that the change has implications on the safety of its users. The *requested* attribute indicates that the change was requested by a stakeholder, but is not of critical importance to the organization or software. If the requested attribute is selected, then none of the other attributes will be selected. This feature is provided because not all change requests are of critical importance to the system. The developer can use more than one of the other attributes to indicate the criticality of the change request when the **requested** attribute is not selected.

5.2.4. Developer experience

The *developer experience* characteristic (Table 10), provides a way to assess how well the developer(s) implementing the change request understand the system architecture [49,92,94,95,110]. The experience of the software development personnel is important factor in change difficulty and in effort prediction. A *minimal* rating indicates that the developer has little experience with the architecture and the components related to the change. A *localized* rating indicates that the developer is experienced with the subset of the architecture related to the change, but not with the entire architecture. Finally, an *extensive* rating indicates that the developer is deeply familiar with the entire architecture [49].

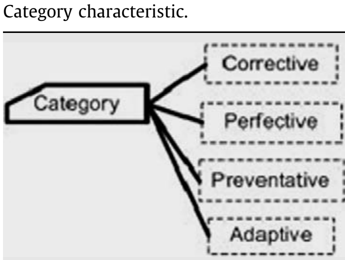
5.2.5. Category

The *category* characteristic (Table 11) classifies the type of change as *perfective*, *corrective*, *adaptive* or *preventative* (described in Section 2). Recording the change category is important for several reasons. The frequency of change types can provide developers with insight about the evolution of the system. For example, Swanson indicates that frequent adaptive changes may be a reflection on system portability. Conversely, frequent perfective changes may indicate a more mature system where maintainability is improved [133]. These conclusions may not be true for all systems in all organizations. But, as a history of changes develops, organization-specific insights can arise.

5.2.6. Granular effect

The *granular effect* describes the extent to which the change affects the architecture (Table 12). *Functional/module* changes affect user-observable attributes and functions of the system. These changes are within a single module. *Subsystem* changes have both a functional and an architectural impact because they affect user functions across modules. Purely *architectural* changes affect only the architecture and not user-observable functions [108]. Architectural changes, which often take place to satisfy a quality attribute

Table 11
Category characteristic.



Corrective [2,4,6,19,30,40,50,66,69,90,91,103,105, 106,119,126,128,133] – subset attributes: intensive evolution
Perfective [2,4,6,19,30,40,50,66,69,90,103–106,119,126,128,133] – Subset attributes: performative, groomative, reductive, enhansive, anticipative, evolutive, design evolution
Preventative [4,6,19,30,50,66,103–106,119,128]
Adaptive [2,6,19,30,40,50,66,69,90,91,103–106,119,126,128,133] – subset attributes: extensive evolution

or non-functional requirement, are often referred to as refactoring or restructuring [27]. *System of system* changes are large-scale changes that affect interactions and architectural components between disparate systems.

5.2.7. Properties

The *properties* characteristic determines the impact of the change on the logical and runtime structures (Table 13). A *static* change affects logical properties, such as module decomposition, module dependency, the inheritance structure and other static properties. A *dynamic* changes affects how data propagation, the behavior of distributed components, execution of concurrent processes, and other runtime behaviors.

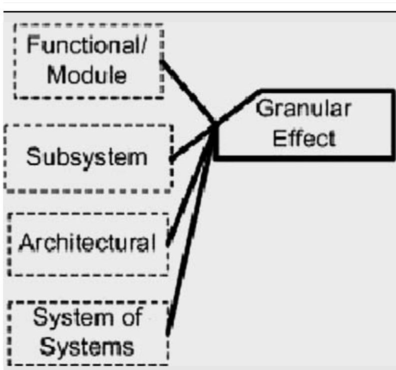
5.2.8. Features

The *features* characteristic (Table 14) determines how the change request will affect the functional requirements of the system. The characteristic identifies impacts on the following areas of the system [18,22,30,94,105,122,124]:

- *devices*: hardware devices used by the system;
- *data access*: receipt of data from external systems/ repositories;
- *data transfer*: flow of data from system to external systems;
- *system interface*: software interfaces with external systems;
- *user interface*: human–computer interaction interfaces;
- *communication*: protocols used to interface other systems/ data;
- *computation*: algorithm functions and modification of data;
- *input/output*: format of information processed by system.

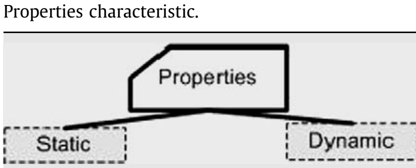
It is expected that the design of a system will determine which aspects are affected by a change [41]. After analyzing the change request to determine which of the above system features will be impacted by the change, the architecture modules that handle those system features must be identified during change analysis. This analysis helps developers and testers focus their effort in the right place.

Table 12
Granular effect characteristic.



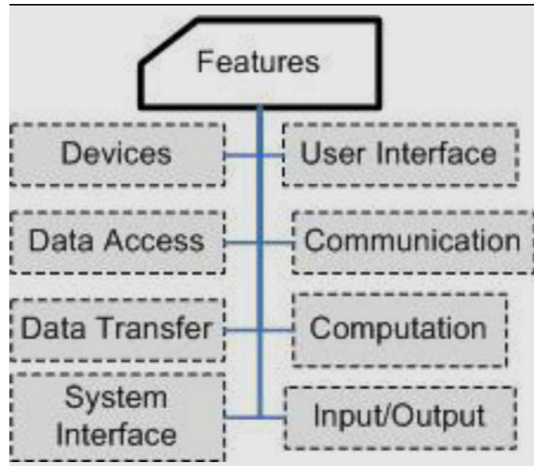
Functional/module [98,108,134]
Subsystem [102] – subset attributes: micro-architecture changes
Architectural [55,67,101,140] – subset attributes: restructuring, refactoring, architecturally significant change, structural changes, macro-architecture changes
System of systems [102]

Table 13
Properties characteristic.



Static [52,58,60,143]
Dynamic [52,58,60]

Table 14
Features characteristic.



Devices [18,30,122,124]
Data access [98]
Data transfer [18,124]
System interface [18,124]
User interface [18,124]
Communication [18,124]
Computation [18,124]
Input/output [18,124]

5.2.9. Quality attributes

The *quality attributes* (Table 15) are areas that are impacted when the change addresses a software quality attribute. The list includes the six quality from ISO Standard 9126 (i.e., *usability*, *reliability*, *functionality*, *portability*, *maintainability*, and *efficiency*) plus additional attributes identified by Krutchen (*availability* and *scalability*) [3,75]. Evaluation of the architecture is critical when addressing a change that focuses on a non-functional (quality) attribute because the architecture may determine whether the goal can be met. [23].

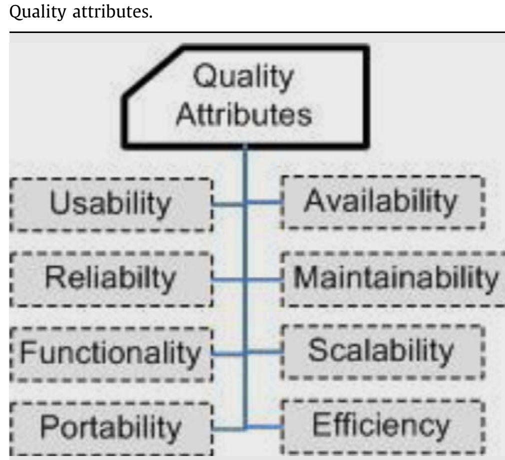
5.2.10. Logical

The *logical* characteristic (Table 16) includes the general architectural features that can be used to describe the static framework of most object-oriented software. These characteristics include: *dependency relationships*, *layers*, *inheritance structure*, *module decomposition*, and *source structure*. Once a logical area is identified as being important (based on its rating on the Overall Impact Scale), the change should be characterized in more detail using the related *Specific Characterization* framework, which is described in Section 5.3.

5.2.11. Runtime

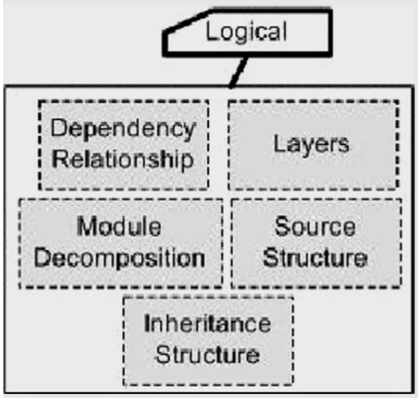
The *runtime* characteristic (Table 17) lists the dynamic architecture attributes common to most object-oriented architectures.

Table 15
Quality attributes.



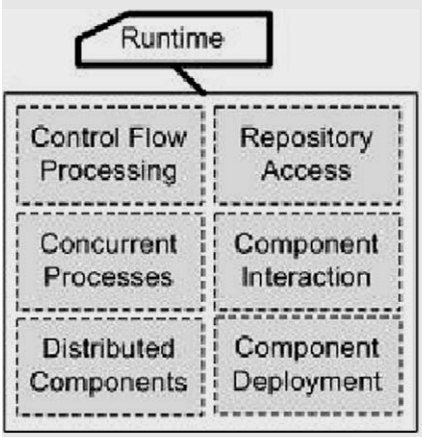
Usability [3]
Reliability [3]
Functionality [3]
Portability [3]
Availability [75]
Maintainability [3]
Scalability [75]
Efficiency [3] – subset attributes: performance change

Table 16
Logical characteristic.



	Dependency relationship [14,45,127] Layers [14,45,127] Module decomposition [14,45,127] – subset attributes: coupling between modules Source structure [14,45,127] – subset attributes: header file changes Inheritance structure [14,45,127] – subset attributes: inheritance deviation

Table 17
Runtime characteristic.



	Control flow processing [14,45,127] Concurrent processes [14,45,127] Distributed components [14,45,127] Repository access [14,45,127] Component interaction [14,45,127] Component deployment [14,45,127]

These characteristics include: *control flow processing*, *repository access*, *concurrent processes*, *component interaction*, *distributed components*, and *component deployment*. Similar to the *logical* characteristic, once a runtime area is identified as being important, the change should be characterized in more detail using the related characteristic from the *Specific Characterization* framework.

5.2.12. Complete general characterization scheme

Using all of the general characteristics described in the previous subsections, we developed the overall characterization scheme shown in Fig. 2. In this figure, the arrows dictate the order of characterization that should be followed. The specific characterization, which allows the developer to provide more detail about the effect of the change on the logical and runtimes structures, is described in the next section.

5.3. Specific characterization

The purpose of these characteristics is to allow the developer to analyze the architecture in more detail to determine how to implement the change. The Specific Impact Scale found in Table 18 describes the magnitude of the changes that can be made to the various architectural structures.

The logical and runtime characteristics focus on static and dynamic relationships among architectural elements. The goal of the specific characterization scheme is to indicate, in a comprehensive manner, which portions of an object-oriented architecture are affected when implementing a change [9,28,31,62,118,123,141,142].

The logical and runtime characteristics selected during the General Characteristics analysis are further elaborated for the Specific Characteristics. For Fig. 3 and 4, the developer would select values from the Specific Impact Scale that correspond to the level of change necessary represented by the oval shapes in the figure. For example, if a change request requires changes to a component within a layer and the addition of a new layer, the developer would select a value from the Specific Impact Scale for the '(Add, Remove, Modify) Layer' and '(Add, Remove, Modify) Layer Module' list of change actions for the 'Layers' view.

5.3.1. Logical views

The *logical* characteristics describe the types of changes that can be made to elements of any view that exhibits those characteristics. Fig. 3 provides a visual overview of these characteristics along with the types of changes that can be made. These changes include adding, modifying, and removing elements and/or the connections between them.

The *Dependency Relationships* view describes the system modules and the relations between them. The *Layers* view abstracts how the system is divided into hierarchical layers. The *Inheritance Structure* view depicts the relationship between the modules in terms of their parent–child–sibling relationships. The *Module Decomposition* view is the basic view of the system at varying levels of abstraction. The *Source Structure* view provides the representation of the location of the source code on the folder.

5.3.2. Runtime views

The *runtime* characteristics describe changes that can be made to portions of the architecture that describe the dynamic aspects of the software. These views contain executable components and connections between those components. The types of changes that can be made to different parts of the architecture are shown in Fig. 4.

The *Control Flow Processing* view shows how system processes interact through a pipe-and-filter representation of the architecture. The *Repository Access* view shows the system in terms of its database and accessor relationship. The *Concurrent Processes* view shows the way processes interact as system threads while the *Component Interaction* view shows processes interaction through the sharing of information through a publish-and-subscribe architecture view. The *Distributed Components* view shows how remote processes interact and the *Component Deployment* view shows the components and their location on system hardware.

5.4. Excluded attributes

For the sake of completeness, Table 19 lists the characteristics that were identified in the literature but not included in the SACCS. The attributes are organized relative to the particular step described in Table 5 that resulted in their exclusion.

6. Research implications and conclusions

This section summarizes the contributions of this review for the software engineering research community. This systematic review of software change provided comprehensive insight into the architecture change process. Several questions were answered regarding changes that affect software architectures. The answers to these questions were derived from peer-reviewed literature sources. These answers also provided the basis for the creation of

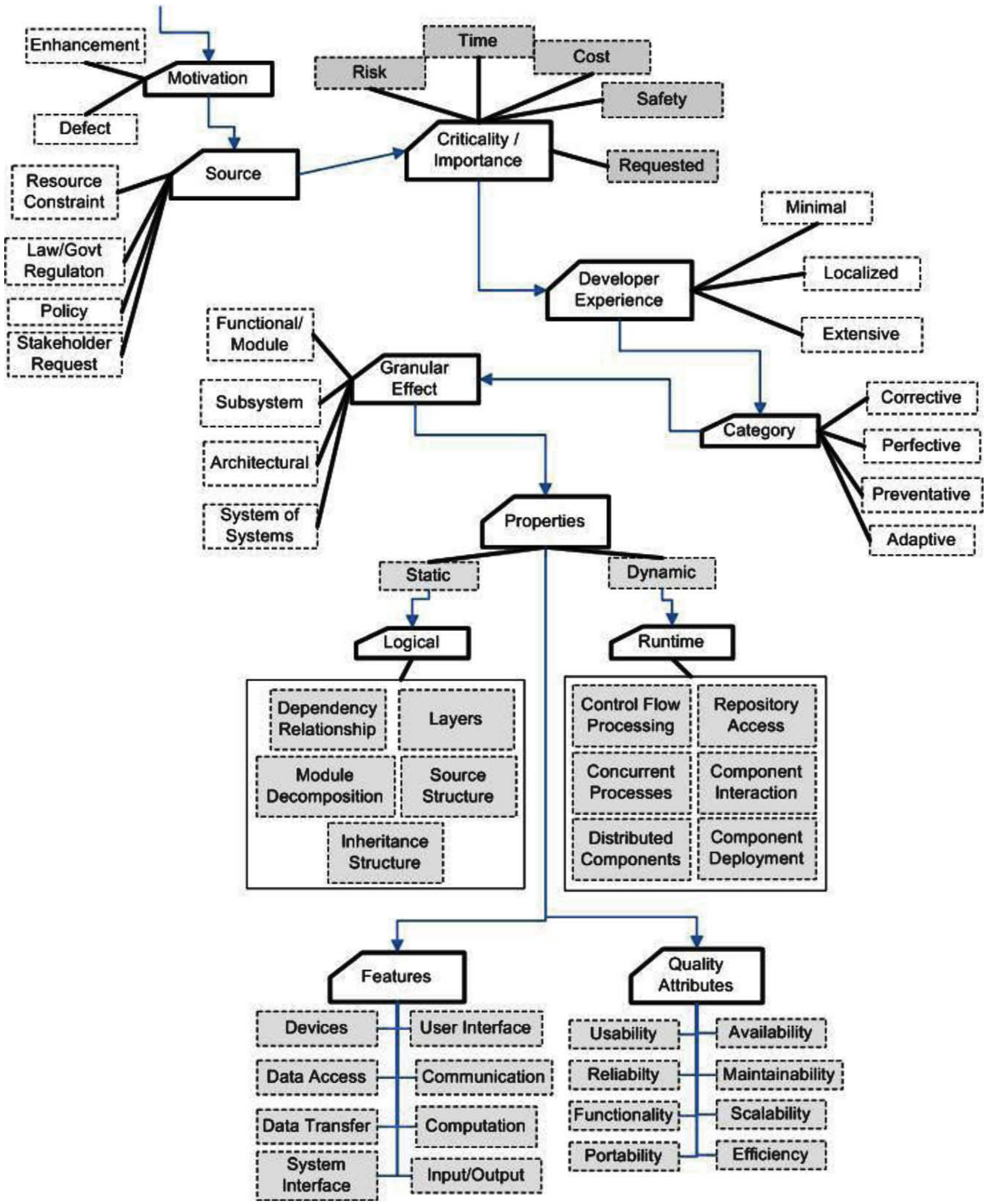


Fig. 2. General Characteristics

SACCS, a framework for assessing change characteristics and their impact to a system.

The systematic review results contained attributes extracted from change classification schemes, change impact analysis techniques, and architectural styles that must be updated to reflect system changes. The results provided a rich set of data that was used to create a comprehensive change characterization scheme. This scheme differs from change classification schemes because it does not lump change requests into a particular class, but provides a way to characterize the change's impact with respect to number of important characteristics based on the experience of the developer. SACCS was created by examining the results of the systematic literature review.

The strengths of this research approach are derived from the systematic process used to perform this initial review. The protocol created prior to conducting the review ensures the completeness of the review. Multiple data sources were used to extract relevant papers. The data sources included relevant journals, conference proceedings, and technical reports. The data extraction form created to obtain consistent data from each of the primary studies, ensured that all relevant information was received from each study.

Table 18
Specific impact scale [79,80].

Rating	Name
0	No impact
1	Small impact – single module/component
2	Small impact – multiple modules/components
3	Significant impact – single module/component
4	Significant impact – multiple modules/components

A major goal for this research is to provide support for system developers and maintainers to assess the potential impact of a proposed change and decide whether it is feasible to implement the change. In cases where the change is crucial to the system, the scheme will help generate consensus on how to approach change implementation and provide an indication of the difficulty. SACCS has been designed as a decision tree where choices made for the high-level characteristics affect decisions that can be made at the more detailed level. The specifics of the relationships among the attributes of the scheme will evolve as additional constraints and dependencies are identified. The characterizations are made using an electronic form. A developer will use this form to record his or her selections individually along with a rationale for the selections. The developer's characterization of the change can then be used to facilitate a discussion among other developers about the proposed impact. The goal is to determine whether the change can be made, given development constraints and architectural complexity.

This systematic review serves a starting point and provides the preliminary framework for a model of application-dependent change difficulty prediction for a change decision support system. The decision support system will take a change request as input

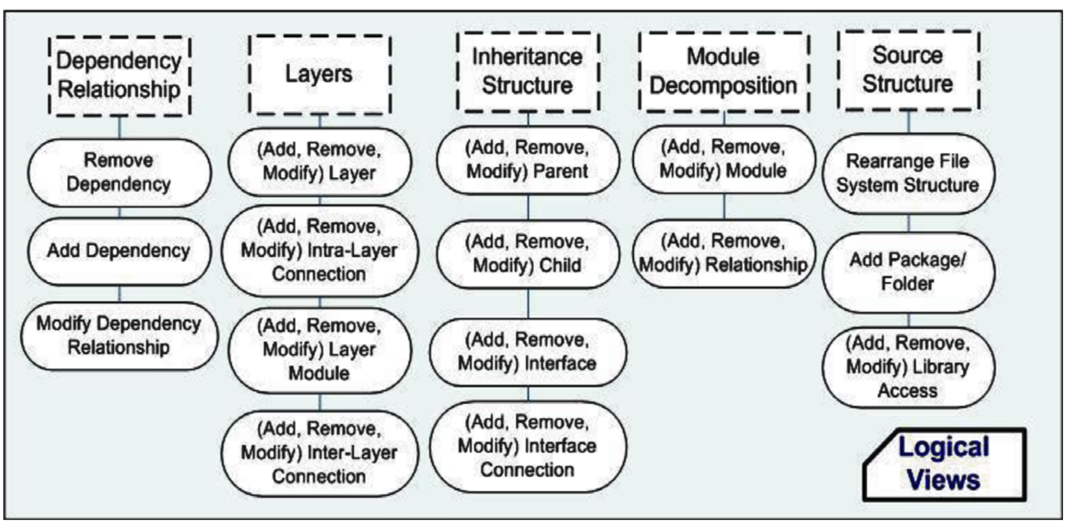


Fig. 3. Logical Views

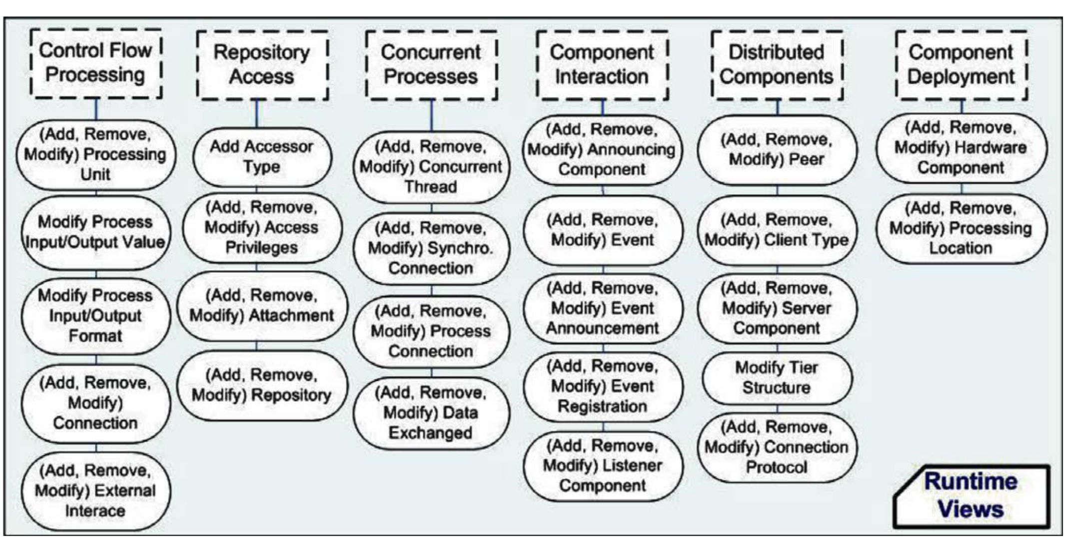


Fig. 4. Runtime Views

Table 19
Excluded characteristics/attributes.

Steps	Title	Attributes
2	Not directly relevant to focus on software architecture	documentation, function name change, parameter change, orderings change, [fields, methods, classes] modification, variable scope change, modifier change, attribute change, variable change, semantic change
3	Not relevant to scope and goals of study	prototype change, coupling between classes, steady-state development, pending changes, replacement changes, slowly developing changes, substitution change, configuration file changes, temporal change properties, design unit change, evolutionary couplings, [kidnapping, splitting, relocating] modules, causal dependency change, ordering dependency change
4	No extensive literature backing	evaluative, consultive, training, updating reformative, retrenchment, retrieving, prettyprinting, user support change, duplicated code change, reposition a code fragment, temporarily adding code fragment, superposition change, direct impact changes and indirect impact changes
5	Proposed outcome of study	coupling understandability, coupling complexity, change process improvement, feedback loop, determining change consequences, statistical change impact

Table A1
Search strings.

String	Name	Search string	Review question (s)	Purpose
1	Software change classification	(software Or application Or system Or module Or component) <And> (change Or refinement Or modification Or revision Or evolution Or modify) <And> (classification Or categorization Or taxonomy Or framework) <And> (scheme Or notation Or chart Or strategy Or schema Or representation Or pattern) <And> **software engineering Or software maintenance**	Q1	To identify the current research being performed in the area of software change classification
2	Software architecture change	(software Or application Or system Or module Or component) <And> (architecture Or high-level design Or conceptual design Or abstract design Or architectural) <And> (change Or refinement Or modification Or revision Or evolution Or modify) <And> (**software architecture Or software maintenance Or software change)**	Q2, Q3, Q5	To determine how software architectures are affected when a software system is changed
3	Software change impact analysis	(software Or application Or system Or module) <And> (change Or refinement Or modification Or revision Or evolution Or modify) <And> (impact Or effect) <And> (analysis Or study Or examination) <And> (**software engineering Or software maintenance Or software change Or impact analysis)**	Q4	To identify and measure change impact related to software systems

and output predictions about what is expected. The ideal implementation of the decision support system is described below.

1. Stakeholder issues change request.
2. Developer formalizes request using template.
3. Developer isolates and characterizes change using SACCS.
4. Proposed changes discussed between project group.
5. Planned implementation recorded.
6. System compares current change to set of previous similar changes.
7. System provides difficulty prediction and analysis.
8. Actual change detail and SACCS characteristics recorded for future use.

The type of questions that the output of the decision support system will provide include: What amount of relative effort is required to implement the change? After the change is isolated to a module, what other historically coupled modules are likely to be affected? Will the change result in a more complex architecture? Will the change violate any architecture constraints? Will the changes be made to historically effort-intensive modules? Should a part of the system with a high handling rate be refactored? Answers to questions such as these will assist a developer to make quality decisions concerning the software project from relevant, application-dependent data.

The results of this research provide a piece of the framework for a data driven change support model that analyzes relevant change data to provide a developer with the tools needed to maintain today's complex software systems. This unique approach that will blend quantitative data analysis and qualitative characterizations that benefit from developer experience is key to this larger system.

SACCS will be refined based upon further analysis and assessment in empirical studies using the scheme. Historical changes that include implementation detail that can be used for validation will be characterized using SACCS. This activity will allow us to identify trends about change characteristics in a particular system and recommend best-practices for future changes with similar characteristics.

Change characterization can be a useful tool in determining the impact of a change. After further research, we envision that this characterization mechanism will be incorporated into an organization's change implementation process. An additional step would be added after receiving a change request to allow the developers to characterize that change request.

Being able to accurately identify changes that will affect software architecture will aid developers in understanding the impact of the change and help them make changes without degrading the quality of the system. We have not found any other change framework that takes a comprehensive approach to change understanding and analysis.

Acknowledgements

We would like to thank Dr. Edward Allen for reviewing this document. We would also like to thank the Empirical Software Engineering Research Group at Mississippi State University for their help and support throughout this research. This research is funded by NSF Grant CCF-0438923.

Appendix A

Search strings were created to extract data from each database. The search strings included terms from each of the research questions and meaningful synonyms and alternate spellings. Table A1 lists the three search strings used to conduct this review, the research question that each string addresses, and the purpose for the search.

Appendix B

This appendix provides a list of all the papers (130) that were included in the review. There were references that were included in the review (i.e., papers whose titles, abstract, content, and quality fit), but they were not cited in the paper. The citations were left out of the paper because some papers provided redundant information to those that were already cited and did not add anything to the review. The references highlighted with a '*' are included in the reference list for the main paper.

[1]* W. Abdelmoez, M. Shereshevsky, R. Gunnalan, H.H. Ammar, Y. Bo, S. Bogazzi, M. Korkmaz, A. Mili, Quantifying Software Architectures: An Analysis of Change Propagation Probabilities, in: The 3rd ACS/IEEE International Conference on Computer Systems and Applications, 2005, pp. 124–131.

[2]* S. Ajila, Software Maintenance: An Approach to Impact Analysis of Objects Change, Software – Practice and Experience, 25(10) (1995) 1155–1181.

[3] V. Ambriola, A. Kmiecik, Architectural Transformations, in: Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering, Ischia, Italy, ACM Press, 2002, pp. 275–278.

[4]* G. Antoniol, G. Canfora, A. De Lucia, Estimating the size of changes for evolving object-oriented systems: a case study, in: Proceedings of the Sixth International Software Metrics Symposium, Boca Raton, FL, 1999, pp. 250–258.

[5]* M. Aoyama, Evolutionary Patterns of Design and Design Patterns, in: Proceedings of the International Symposium on Principles of Software Evolution, 2000, pp. 110–116.

[6]* M. Aoyama, Metrics and Analysis of Software Architecture Evolution with Discontinuity, in: Proceedings of the International Workshop on Principles of Software Evolution, Orlando, FL, ACM Press, 2002, pp. 103–107.

[7]* T. Apiwattanapong, A. Orso, M.J. Harrold, Efficient and precise dynamic impact analysis using execute-after sequences, in: Proceedings of the 27th International Conference on Software Engineering, St. Louis, MO, 2005, pp. 432–441.

[8]* R.S. Arnold, S.A. Bohner, Impact analysis-towards a framework for comparison, in: Proceeding of the Conference on Software Maintenance, Montreal, Que., 1993, pp. 292–301.

[9]* R.D. Banker, S.M. Datar, C.F. Kemerer, D. Zweig, Software complexity and maintenance costs, Communications of the ACM, 36(11) (1993) 81–94.

[10]* J. Baragry, K. Reed, Why we need a different view of software architecture, in: Proceedings of the Working IEEE/IFIP Conference on Software Architecture, Amsterdam, 2001, pp. 125–134.

[11]* O. Barais, L. Duchien, A.F. Le Meur, A Framework to Specify Incremental Software Architecture Transformations, in: 31st EUROMICRO Conference on Software Engineering and Advanced Applications 2005, Porto, Portugal, pp. 62–69.

[12]* E. Barry, C.F. Kemerer, S.A. Slaughter, On the Uniformity of Software Evolution Patterns, in: Proceedings of the International Conference on Software Engineering, Portland, OR, 2003, pp. 106–113.

[13]* E. Barry, S. Slaughter, C.F. Kemerer, An empirical analysis of software evolution profiles and outcomes, in: Proceeding of the 20th International Conference on Information Systems, Charlotte, NC, Association for Information Systems, 1999, pp. 453–458.

[14]* V. Basili, L. Briand, S. Condon, Y. Kim, W. Melo, J. Valen, Understanding and predicting the process of software maintenance releases, in: Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany, 1996, pp. 464–474.

[15]* V. Basili, B.T. Perricone, Software errors and complexity: an empirical investigation, Communications of the ACM, 27 (1) (1984) 42–52.

[16]* V. Basili, D. Weiss, Evaluation of a software requirements document by analysis of change data, in: Proceedings of the 5th International Conference on Software Engineering, IEEE Press, San Diego, CA, 1981, pp. 314–323.

[17]* P. Bengtsson, J. Bosch, Architecture level prediction of software maintenance, in: Proceedings of 3rd EuroMicro Conference on Maintenance and Reengineering (CSMR'99), Amsterdam, Netherlands, 1999, pp. 139–147.

[18]* S.A. Bohner, Impact analysis in the software change process: a year 2000 perspective, in: Proceedings of the International Conference on Software Maintenance, Monterey, CA, 1996, pp. 42–51.

[19]* S.A. Bohner, Software change impacts-an evolving perspective, in: Proceedings of the International Conference on Software Maintenance, Montreal, Quebec, Canada, 2002, pp. 263–272.

[20]* P. Bose, Change analysis in an architectural model: a design rationale based approach, in: Proceedings of the Third International Workshop on Software Architecture, ACM Press, Orlando, FL, 1998, pp. 5–8.

[21]* L. Briand, V. Basili, A classification procedure for the effective management of changes during the maintenance process, in: Proceeding of the Conference on Software Maintenance, Orlando, FL, 1992, pp. 328–336.

[22]* L. Briand, Y. Labiche, L. O'sullivan, Impact analysis and change management of Uml models, in: Proceedings of the International Conference on Software Maintenance, Amsterdam, The Netherlands, 2003, pp. 256–265.

[23]* L. Briand, Y. Labiche, L. O'sullivan, M.M. Sûwka, Automated impact analysis of Uml models, Journal of Systems and Software, 79(3) (2006) 339–352.

[24]* L. Briand, S. Morasca, V. Basili, Measuring and assessing maintainability at the end of high level design, in: Proceedings of the Conference on Software Maintenance, Montreal, Que., Canada, 1993, pp. 88–97.

[25]* L. Briand, J. Wust, H. Lounis, Using coupling measurement for impact analysis in object-oriented systems, in: Proceedings of the Conference on Software Maintenance, Oxford, 1999, pp. 475–482.

[26]* J. Buckley, T. Mens, M. Zenger, A. Rashid, G. Kniesel, Towards a taxonomy of software change, Journal of Software Maintenance and Evolution: Research and Practice, 17(5) (2005) 309–332.

[27]* A. Capiluppi, A.E. Faria, J.F. Ramil, Exploring the relationship between cumulative change and complexity in an open source system, in: Ninth European Conference on Software Maintenance and Reengineering, King's College, London, 2005, pp. 21–29.

[28]* A. Capiluppi, J. Fernandez-Ramil, J. Higman, H.C. Sharp, N. Smith, An empirical study of the evolution of an agile-developed software system, in: 29th International Conference on Software Engineering, Minneapolis, MN, 2007, pp. 511–518.

[29]* S.J. Carriere, R. Kazman, S.G. Woods, Assessing and maintaining architectural quality, in: Proceedings of the Third European Conference on Software Maintenance and Reengineering, Amsterdam, 1999, pp. 22–30.

[30] N. Chapin, Software maintenance types-a fresh view, in: Proceedings of the International Conference on Software Maintenance, 2000, pp. 247–252.

[31]* N. Chapin, J.E. Hale, K.M. Khan, J.F. Ramil, W.-G. Tan, Types of software evolution and software maintenance, Journal of Software Maintenance: Research and Practice 13(1) (2001) pp. 3–30.

[32]* M.A. Chaumun, H. Kabaili, R.K. Keller, F. Lustman, A change impact model for changeability assessment in object-oriented software systems, in: Proceedings of the Third European Conference on Software Maintenance and Reengineering, 1999, pp. 130–138.

[33]* M.A. Chaumun, H. Kabaili, R.K. Keller, F. Lustman, A change impact model for changeability assessment in object-oriented software systems, Science of Computer Programming 45(2) (2002) 155–177.

[34]* O.C. Chesley, X. Ren, B.G. Ryder, Crisp: a debugging tool for Java programs, in: Proceedings of the International Conference on Software Maintenance, 2005, pp. 401–410.

[35]* P. Clarke, B. Malloy, P. Gibson, Using a taxonomy tool to identify changes in Oo software, in: Proceedings of the Seventh European Conference on Software Maintenance and Reengineering, 2003, pp. 213–222.

[36] R. Conradi, M.N. Nguyen, A.I. Wang, Planning support to software process evolution, International Journal of Software Engineering and Knowledge Engineering, 10(1) (2000) 31–47.

[37]* S. Cook, R. Harrison, M.M. Lehman, P. Wernick, Evolution in software systems: foundations of the spe classification scheme, Journal of Software Maintenance and Evolution: Research and Practice, 18(1) (2005) 1–35.

[38]* S. Cook, R. Harrison, P. Wernick, A simulation model of self-organising evolvability in software systems, in: IEEE International Workshop on Software Evolvability, 2005, pp. 17–22.

[39]* R.T. Crocker, A. Von Mayrhauser, Maintenance support needs for object-oriented software, in: Proceedings of the Seventeenth Annual International Computer Software and Applications Conference, 1993, Phoenix, AZ, pp. 63–69.

[40]* S. Dekleva, Delphi study of software maintenance problems, in: Proceedings of the Conference on Software Maintenance, 1992, Orlando, FL, pp. 10–17.

[41]* S.G. Eick, T.L. Graves, A.F. Karr, J.S. Marron, A. Mockus, Does code decay? assessing the evidence from change management data, IEEE Transactions on Software Engineering, 27(1) (2001) 1–12.

[42]* S.G. Eick, T.L. Graves, A.F. Karr, A. Mockus, P. Schuster, Visualizing software changes, Software Engineering, IEEE Transactions on, 28(4) (2002) 396–412.

[43] K. Falkner, H. Detmold, D. Howard, D.S. Munro, R. Morrison, S. Norcross, Unifying static and dynamic approaches to evolution through the compliant systems architecture, in: Proceedings of the 37th Annual Hawaii International Conference on System Sciences, 2004, Big Island, HI, pp. 268.

[44]* T. Feng, J.I. Maletic, Applying dynamic change impact analysis in component-based architecture design, in: Seventh ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2006, pp. 43–48.

[45]* R. Fiutem, P. Tonella, G. Antoniol, E. Merlo, A cliche-based environment to support architectural reverse engineering, in Proceedings of the Third Working Conference on Reverse Engineering, 1996, pp. 319–328.

[46]* B. Fluri, H.C. Gall, Classifying change types for qualifying change couplings, in: Proceedings of the 14th IEEE Conference on Program Comprehension, 2006, Athens, Greece, pp. 35–45.

[47]* E. Fuentetaja, D.J. Bagert, Software evolution from a time-series perspective, in: Proceedings of the International Conference on Software Maintenance, 2002, pp. 226–229.

[48]* H. Gall, M. Jazayeri, R.R. Klosch, G. Trausmuth, Software evolution observations based on product release history, in: Proceedings of the International Conference on Software Maintenance, 1997, pp. 160–166.

[49]* J. Grundy, J. Hosking, High-level static and dynamic visualisation of software architectures, in: Proceedings of the 2000 IEEE International Symposium on Visual Languages, 2000, Seattle, WA, pp. 5–12.

[50]* A. Hac, Software renovation for large software applications, in: IEEE Region 10 International Conference (TENCON '92), 1992, Melbourne, Australia, pp. 307–311.

[51]* G.T. Heineman, A. Mehtra, Architectural evolution of legacy systems, in: Proceedings of the 23rd Annual International Computer Software and Applications Conference, 1999, Phoenix, AZ, pp. 4–12.

[52]* I. Herraiz, G. Robles, J.M. Gonzalez-Barahona, A. Capiluppi, J.F. Ramil, Comparison between slocs and number of files as size metrics for software evolution analysis, in: Proceedings of the 10th European Conference on Software Maintenance and Reengineering 2006, 8pp.

[53]* D.S. Hinley, Software evolution management: a process-oriented perspective, Information and Software Technology 38(11) (1996) 723–730.

[54]* L. Hochstein, M. Lindvall, Combating architectural degeneration: a survey, Information and Software Technology, 47(10) (2005) 643–656.

[55]* P. Hsia, A. Gupta, C. Kung, J. Peng, S. Liu, A study on the effect of architecture on maintainability of object-oriented systems, in: Proceedings of the International Conference on Software Maintenance, 1995, pp. 4–11.

[56]* I. Ivkovic, K. Kontogiannis, A framework for software architecture refactoring using model transformations and semantic annotations, in: Proceedings of the Conference on Software Maintenance and Reengineering, 2006, pp. 135–144.

[57] M. Jørgensen, An empirical study of software maintenance tasks, Journal of Software Maintenance: Research and Practice, 7(1) (1995) 27–48.

[58]* C.F. Kemerer, S. Slaughter, An empirical approach to studying software evolution, IEEE Transactions on Software Engineering, 25(4) (1999) 493–503.

[59]* S. Kim, E.J. Whitehead, J. Bevan, Analysis of signature change patterns, in: Proceedings of the 2005 International Workshop on Mining Software Repositories, 2005, St. Louis, MO, ACM Press, pp. 1–5.

[60]* S. Kim, E.J. Whitehead, Y. Zhang, Classifying software changes: clean or buggy? IEEE Transactions on Software Engineering, 34(2) (2008) 181–196.

[61]* B.A. Kitchenham, G.H. Travassos, A.V. Mayrhauser, F. Niessink, N.F. Schneidewind, J. Singer, S. Takada, R. Vehvilainen, H. Yang, Towards an ontology of software maintenance, Journal of Software Maintenance: Research and Practice, 11(6) (1999) 365–389.

[62]* P. Kruchten, Architectural Blueprints—the "4 + 1 view model of software architecture, IEEE Software, 12(6) (1995) 42–50.

[63]* D. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, C. Chen, Change impact identification in object oriented software maintenance, in: Proceedings of the International Conference on Software Maintenance, 1994, Victoria, BC, pp. 202–211.

[64]* W. Lam, V. Shankararaman, Managing change in software development using a process improvement approach, in: Proceedings of the 24th Annual Euromicro Conference, 1998, Vasteras, pp. 779–786.

[65]* W. Lam, V. Shankararaman, Requirements change: a dissection of management issues, in: Proceedings of the 25th EuroMicro Conference, 1999, Milan, Italy, pp. 244–251.

[66]* N. Lassing, D. Rijsenbrij, H. Van Vliet, Flexibility of the combad architecture, in: Proceedings of the First Working IFIP Conference on Software Architecture, 1999, pp. 357–368.

[67]* N. Lassing, D. Rijsenbrij, H. Van Vliet, Towards a broader view on software architecture analysis of flexibility, in: Proceedings of the Sixth Asia Pacific Conference on Software Engineering, 1999, pp. 238–245.

[68]* Y. Lee, J. Yang, K.H. Chang, Metrics and evolution in open source software, in: Seventh International Conference on Quality Software, 2007, pp. 191–197.

[69]* M.M. Lehman, Programs, life cycles, and laws of software evolution, Proceedings of the IEEE, 68(9) (1980) 1060–1076.

[70]* M.M. Lehman, Feedback, evolution and software technology, in: Proceedings of the 10th International Process Support of Software Product Lines Software Process Workshop, 1996, pp. 101–103.

[71]* M.M. Lehman, D.E. Perry, J.F. Ramil, Implications of evolution metrics on software maintenance, in: Proceedings of the International Conference on Software Maintenance, 1998, Bethesda, MD, pp. 208–217.

[72]* M.M. Lehman, D.E. Perry, J.F. Ramil, On evidence supporting the feast hypothesis and the laws of software evolution, in: Proceedings of the Fifth International Software Metrics Symposium, 1998, pp. 84–88.

[73]* M.M. Lehman, J.F. Ramil, Towards a theory of software evolution –and its practical impact, in: Proceedings of the International Symposium on Principles of Software Evolution, 2000, pp. 2–1.

[74]* M.M. Lehman, J.F. Ramil, P.D. Wernick, D.E. Perry, W.M. Turski, metrics and laws of software evolution-the nineties view, in: Proceedings of the Fourth International Software Metrics Symposium, 1997, pp. 20–32.

[75]* I.H. Lin, D.A. Gustafson, Classifying software maintenance, in: Proceedings of the Conference on Software Maintenance, 1988, Scottsdale, AZ, pp. 241–247.

[76] M. Lindvall, Monitoring and measuring the change-prediction process at different granularity levels: an empirical study, Software Process: Improvement and Practice, 4(1) (1998) 3–10.

[77]* M. Lindvall, K. Sandahl, How well do experienced software developers predict software change? Journal of Systems and Software, 43(1) (1998) 19–27.

[78]* M. Lindvall, R. Tesoriero, P. Costa, Avoiding architectural degeneration: an evaluation process for software architecture, in: Proceedings of the Eighth IEEE Symposium on Software Metrics, 2002, pp. 77–86.

[79] X. Liu, Q. Wang, Study on application of a quantitative evaluation approach for software architecture adaptability, in: Proceedings of the Fifth International Conference on Quality Software 2005, Melbourne, Australia, pp. 265–272.

[80]* N.H. Madhavji, The prism model of changes, in: Proceedings of the 13th International Conference on Software Engineering 1991, Austin, TX, pp. 166–177.

[81]* N.H. Madhavji, Environment evolution: the prism model of changes, IEEE Transactions on Software Engineering, 18(5) (1992) 380–392.

[82]* J.F. Maranzano, S.A. Rozsypal, G.H. Zimmerman, G.W. Warnken, P.E. Wirth, D.M. Weiss, Architecture reviews: practice and experience, IEEE Software, 22(2) (2005) 34–43.

[83]* P.J. Mayhew, C.J. Worsley, P.A. Dearnley, Control of software prototyping process: change classification approach, Information and Software Technology, 31(2) (1989) 59–66.

[84]* D.S. Mccrickard, G.D. Abowd, Assessing the impact of changes at the architectural level: a case study on graphical debuggers, in: Proceedings of the International Conference on Software Maintenance 1996, Monterey, CA, pp. 59–67.

[85]* T. Mens, J. Buckley, M. Zenger, A. Rashid, Towards a taxonomy of software evolution, in: Proceedings of the International Workshop on Unanticipated Software Evolution, 2003, Warsaw, Poland, pp. 1–18.

[86] T. Mens, T. Tourwe, A survey of software refactoring, IEEE Transactions on Software Engineering, 30(2) (2004) 126–139.

[87]* R. Mittermeir, Software evolution: let's sharpen the terminology before sharpening (out-of-scope) tools, in: Proceedings of the 4th International Workshop on Principles of Software Evolution, 2001, Vienna, Austria, ACM Press, pp. 114–121.

[88]* A. Mockus, L.G. Votta, Identifying reasons for software changes using historic databases, in: Proceedings of the International Conference on Software Maintenance, 2000, San Jose, CA, pp. 120–130.

[89]* A. Mockus, D. Weiss, Predicting risk of software changes, Bell Labs Technical Journal, 5(2) (2000) 169–180.

[90]* P. Mohagheghi, R. Conradi, An empirical study of software change: origin, acceptance rate, and functionality vs. quality attributes, in: Proceedings of the 2004 International Symposium on Empirical Software Engineering (ISESE '04), 2004, pp. 7–16.

[91]* V. Nanda, N.H. Madhavji, The impact of environmental evolution on requirements changes, in: Proceedings of the International Conference on Software Maintenance, 2002, pp. 452–461.

[92]* J. Nedstam, E.A. Karlsson, M. Host, The architectural change process, in: Proceedings of the 2004 International Symposium on Empirical Software Engineering (ISESE '04), 2004, pp. 27–36.

[93]* M.N. Nguyen, R. Conradi, Classification of meta-processes and their models, in: Proceedings of the Third International Conference on the Software Process 1994, pp. 167–175.

[94]* N. Nurmuliani, D. Zowghi, S.P. Williams, Using card sorting technique to classify requirements change, in: Proceedings of the 12th IEEE International Requirements Engineering Conference, 2004, pp. 240–248.

[95]* J.S. O'neal, D.L. Carver, Analyzing the impact of changing requirements, in: Proceedings of the International Conference on Software Maintenance, 2001, Florence, pp. 190–195.

[96]* C. O'reilly, P. Morrow, D. Bustard, Lightweight prevention of architectural erosion, in: Proceedings of the Sixth International Workshop on Principles of Software Evolution, 2003, Helsinki, Finland, pp. 59–64.

[97]* D.L. Parnas, Software aging, in: Proceedings of the 16th International Conference on Software Engineering, 1994, Sorrento, Italy, pp. 279–287.

[98]* N. Phillips, S. Black, Distinguishing between learning, growth and evolution, in: IEEE International Workshop on Software Evolvability, 2005, pp. 49–52.

[99]* D. Port, H. Liguo, Strategic architectural flexibility, in: Proceedings of the International Conference on Software Maintenance 2003, Amsterdam, The Netherlands, pp. 389–396.
[100]* A. Postma, P. America, J.G. Wijnstra, Component replacement in a long-living architecture: the 3rdba approach, in: Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA '04), 2004, pp. 89–98.
[101]* J.S. Poulin, Evolution of a software architecture for management information systems, in: Joint Proceedings of the Second International Software Architecture Workshop and International Workshop on Multiple Perspectives in Software Development 1996, San Francisco, CA, ACM Press, pp. 134–137.
[102]* R. Purushothaman, D.E. Perry, Toward understanding the rhetoric of small source code changes, IEEE Transactions on Software Engineering, 31 (6) (2005) 511–526.
[103] V. Rajlich, P. Gosavi, Incremental change in object-oriented programming, IEEE Software, 21(4) (2004) 62–69.
[104]* J.F. Ramil, Laws of software evolution and their empirical support, in: Proceedings of the International Conference on Software Maintenance, 2002, pp. 71–71.
[105]* J. Ratzinger, M. Fischer, H. Gall, Improving evolvability through refactoring, in: Proceedings of the 2005 International Workshop on Mining Software Repositories, 2005, St. Louis, MO, ACM Press, pp. 1–5.
[106]* X. Ren, F. Shah, F. Tip, B.G. Ryder, O. Chesley, Chianti: a tool for change impact analysis of Java programs, in: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications 2004, Vancouver, BC, Canada, ACM Press, pp. 432–448.
[107]* J.E. Robbins, D.M. Hilbert, D.F. Redmiles, Using critics to analyze evolving architectures, in: Joint Proceedings of the Second International Software Architecture Workshop and International Workshop on Multiple Perspectives in Software Development 1996, San Francisco, CA, ACM Press, pp. 90–93.
[108]* H. Rombach, B. Ulery, J. Valett, Toward full life cycle control: adding maintenance measurement to the sel, Journal of Systems and Software, 18(2) (1992) 125–135.
[109]* N. Sadou, D. Tamzalit, M. Oussalah, A unified approach for software architecture evolution at different abstraction levels, in: Proceedings of the 2005 Eighth International Workshop on Principles of Software Evolution, 2005, pp. 65–68.
[110]* S.R. Schach, B. Jin, L. Yu, G.Z. Heller, J. Offutt, Determining the distribution of maintenance categories: survey versus measurement, Empirical Software Engineering, 8(4) (2003) 351–365.
[111] N.F. Schneidewind, Investigation of the risk to software reliability and maintainability of requirements changes, in: Proceedings of the IEEE International Conference on Software Maintenance, 2001, Florence, pp. 127–136.
[112] G.E. Stark, Measurements for managing software maintenance, in: Proceedings of IEEE International Conference on Software Maintenance, 1996, pp. 152–161.
[113]* R.C. Sugden, M.R. Strens, Strategies, tactics and methods for handling change, in: Proceedings of the IEEE Symposium and Workshop on Engineering of Computer-Based Systems, 1996, Friedrichshafen, pp. 457–463.
[114]* M. Svahnberg, C. Wohlin, An investigation of a method for identifying a software architecture candidate with respect to quality attributes, Empirical Software Engineering, 10(2) (2005) 149–181.
[115]* M. Svahnberg, C. Wohlin, L. Lundberg, M. Mattsson, Quality attribute-driven selection of software architecture structures, in: Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering, 2002, Ischia, Italy, ACM Press, pp. 819–826.
[116]* B. Swanson, "The dimensions of maintenance, in: Proceedings of the 2nd International Conference on Software Engineering 1976, San Francisco, CA, IEEE Computer Society Press, pp. 492–497.
[117]* L. Tahvildari, R. Gregory, K. Kontogiannis, An approach for measuring software evolution using source code features, in: Proceedings of the Sixth Asia Pacific Software Engineering Conference, 1999, Takamatsu, pp. 10–17.
[118]* J.B. Tran, R.C. Holt, Forward and reverse repair of software architecture, in: Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research, 1999, Mississauga, Ontario, Canada IBM Press, pp. 12–21.
[119]* Q. Tu, M.W. Godfrey, An integrated approach for studying architectural evolution, in: Proceedings of the 10th International Workshop on Program Comprehension, 2002, pp. 127–136.
[120]* R.T. Tvedt, P. Costa, M. Lindvall, Does the code match the design? a process for architecture evaluation, in: Proceedings of the International Conference on Software Maintenance, 2002, pp. 393–401.
[121]* R.T. Tvedt, M. Lindvall, P. Costa, A process for software architecture evaluation using metrics, in: Proceedings of the 27th Annual NASA Goddard/IEEE Software Engineering Workshop, 2002, pp. 191–196.
[122]* F. Van Rysselberghe, S. Demeyer, Mining version control systems for Facs (Frequently Applied Changes), in: 26th International Conference on Software Engineering, 2004, Edinburgh, Scotland, pp. 48–52.
[123]* P. Weissgerber, S. Diehl, Identifying refactorings from source-code changes, in: 21st IEEE/ACM International Conference on Automated Software Engineering, 2006, pp. 231–240.
[124]* Z. Xing, E. Stroulia, Analyzing the evolutionary history of the logical design of object-oriented software, IEEE Transactions on Software Engineering, 31(10) (2005) 850–868.
[125]* Z. Xing, E. Stroulia, Umldiff: an algorithm for object-oriented design differencing, in: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, 2005, Long Beach, CA, ACM Press
[126]* Y. Yang, C. Riva, Scenarios for mining the software architecture evolution, in: Proceedings of the 2006 International Workshop on Mining Software Repositories, 2006, Shanghai, China, ACM Press, pp. 10–13.
[127] A.T.T. Ying, G.C. Murphy, R. Ng, M.C. Chu-Carroll, Predicting source code changes by mining change history, IEEE Transactions on Software Engineering, 30(9) (2004) 574–586.
[128] J. Zhao, H. Yang, L. Xiang, B. Xu, Change impact analysis to support architectural evolution, Journal of Software Maintenance and Evolution: Research and Practice 14(5) (2002) 317–333.
[129] T. Zimmermann, A. Zeller, P. Weissgerber, S. Diehl, Mining version histories to guide software changes, IEEE Transactions on Software Engineering 31(6) (2005) 429–445.
[130] D. Zowghi, R. Offen, A logical framework for modeling and reasoning about the evolution of requirements, in: Proceedings of the Third IEEE International Symposium on Requirements Engineering, 1997, Annapolis, MD, pp. 247–257.

References

[1] Ieee Standard for a Software Quality Metrics Methodology, IEEE Std 1061-1998, 1998.
[2] IEEE Standard for Software Maintenance, IEEE Std 1219-1998, Institute for Electrical and Electronic Engineer, New York, NY, 1998.
[3] ISO/IEC FCD 9126-1.2 Information Technology––Software Product Quality, ISO/IEC FCD 9126-1.2, 1998.
[4] ISO/IEC. Software Engineering—Software Maintenance Standard, ISO/IEC FDIS 14764, International Standards Organization, Geneva, Switzerland, 1999.
[5] W. Abdelmoez, M. Shereshevsky, R. Gunnalan, H.H. Ammar, Y. Bo, S. Bogazzi, M. Korkmaz, A. Mili, Quantifying software architectures: an analysis of change propagation probabilities, in: The 3rd ACS/IEEE International Conference on Computer Systems and Applications, 2005, pp. 124–131.
[6] S. Ajila, Software maintenance: an approach to impact analysis of objects change, Software – Practice and Experience 25 (10) (1995) 1155–1181.
[7] G. Antoniol, G. Canfora, A. De Lucia, Estimating the size of changes for evolving object-oriented systems: a case study, in: Proceedings of the Sixth International Software Metrics Symposium, Boca Raton, FL, 1999, pp. 250–258.
[8] M. Aoyama, Evolutionary patterns of design and design patterns, in: Proceedings of the International Symposium on Principles of Software Evolution, 2000, pp. 110–116.
[9] M. Aoyama, Metrics and analysis of software architecture evolution with discontinuity, in: Proceedings of the International Workshop on Principles of Software Evolution, ACM Press, Orlando, FL, 2002, pp. 103–107.
[10] T. Apiwattanapong, A. Orso, M.J. Harrold, Efficient and precise dynamic impact analysis using execute-after sequences, in: Proceedings of the 27th International Conference on Software Engineering, St. Louis, MO, 2005, pp. 432–441.
[11] R.S. Arnold, S.A. Bohner, Impact analysis-towards a framework for comparison, in: Proceeding of the Conference on Software Maintenance, Montreal, Que., 1993, pp. 292–301.
[12] J. Arthur, Software Evolution: A Software Maintenance Challenge, John Wiley & Sons, 1988
[13] R.D. Banker, S.M. Datar, C.F. Kemerer, D. Zweig, Software complexity and maintenance costs, Communications of the ACM 36 (11) (1993) 81–94.
[14] J. Baragry, K. Reed, Why we need a different view of software architecture, in: Proceedings of the Working IEEE/IFIP Conference on Software Architecture, Amsterdam, 2001, pp. 125–134.
[15] O. Barais, L. Duchien, A.F. Le Meur, A framework to specify incremental software architecture transformations, in: 31st EUROMICRO Conference on Software Engineering and Advanced Applications 2005, Porto, Portugal, pp. 62–69.
[16] M. Barbacci, M.H. Klein, T.A. Longstaff, C.B. Weinstock, Quality Attributes, CMU/SEI-95-TR-021, Software Engineering Institute, , Carnegie Mellon, 1995.
[17] E. Barry, C.F. Kemerer, S.A. Slaughter, On the uniformity of software evolution patterns, in: Proceedings of the International Conference on Software Engineering, Portland, OR, 2003, pp. 106–113.
[18] E. Barry, S. Slaughter, C.F. Kemerer, An empirical analysis of software evolution profiles and outcomes, in: Proceeding of the 20th International Conference on Information Systems, Association for Information Systems, Charlotte, NC, 1999, pp. 453–458.

[19] V. Basili, L. Briand, S. Condon, Y. Kim, W. Melo, J. Valen, Understanding and predicting the process of software maintenance releases, in: Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany, 1996, pp. 464–474.
[20] V. Basili, G. Caldiera, H.D. Rombach, The goal question metric paradigm, in: J.J. Marciniak (Ed.), Encyclopedia of Software Engineering, John Wiley & Sons, Inc., New York, 1994, pp. 528–532.
[21] V. Basili, B.T. Perricone, Software errors, complexity: an empirical investigation, Communications of the ACM 27 (1) (1984) 42–52.
[22] V. Basili, D. Weiss, Evaluation of a software requirements document by analysis of change data, in: Proceedings of the 5th International Conference on Software Engineering, IEEE Press, San Diego, CA, 1981, pp. 314–323.
[23] P. Bengtsson, J. Bosch, Architecture level prediction of software maintenance, in: Proceedings of 3rd EuroMicro Conference on Maintenance and Reengineering (CSMR'99), Amsterdam, Netherlands, 1999, pp. 139–147.
[24] J. Biolchini, P. Mian, A. Natali, G. Travassos, Systematic Review in Software Engineering, RT – ES 679 / 05, S.E.A.C.S. Department, COPPE/UFRJ, 2005, Rio de Janeiro, Brazil.
[25] S.A. Bohner, Impact analysis in the software change process: a year 2000 perspective, in: Proceedings of the International Conference on Software Maintenance, Monterey, CA, 1996, pp. 42–51.
[26] S.A. Bohner, Software change impacts-an evolving perspective, in: Proceedings of the International Conference on Software Maintenance, Montreal, Quebec, Canada, 2002, pp. 263–272.
[27] J. Bosch, Design and Use of Software Architectures, Addison Wesley, 2000.
[28] P. Bose, Change analysis in an architectural model: a design rationale based approach, in: Proceedings of the Third International Workshop on Software Architecture, ACM Press, Orlando, FL, 1998, pp. 5–8.
[29] B. Brereton, B. Kitchenham, D. Budgen, M. Turner, M. Khalil, Employing Systematic Literature Review: An Experience Report, Technical Report TR 05/01, School of Computing & Mathematics, Keele University, Keele, Staffordshire, UK, 2005.
[30] L. Briand, V. Basili, A classification procedure for the effective management of changes during the maintenance process, in: Proceeding of the Conference on Software Maintenance, Orlando, FL, 1992, pp. 328–336.
[31] L. Briand, Y. Labiche, L. O'sullivan, Impact analysis and change management of Uml models, in: Proceedings of the International Conference on Software Maintenance, Amsterdam, The Netherlands, 2003, pp. 256–265.
[32] L. Briand, Y. Labiche, L. O'sullivan, M.M. Sûwka, Automated impact analysis of Uml models, Journal of Systems, Software 79 (3) (2006) 339–352.
[33] L. Briand, S. Morasca, V. Basili, Measuring and assessing maintainability at the end of high level design, in: Proceedings of the Conference on Software Maintenance, Montreal, Que., Canada, 1993, pp. 88–97.
[34] L. Briand, J. Wust, H. Lounis, Using coupling measurement for impact analysis in object-oriented systems, in: Proceedings of the Conference on Software Maintenance, Oxford, 1999, pp. 475–482.
[35] F. Brooks, The Mythical Man–Month, Addison-Wesley, 1975.
[36] J. Buckley, T. Mens, M. Zenger, A. Rashid, G. Kniesel, Towards a taxonomy of software change, Journal of Software Maintenance, Evolution: Research, Practice 17 (5) (2005) 309–332.
[37] A. Capiluppi, A.E. Faria, J.F. Ramil, Exploring the relationship between cumulative change and complexity in an open source system, in: Ninth European Conference on Software Maintenance and Reengineering, King's College, London, 2005, pp. 21–29.
[38] A. Capiluppi, J. Fernandez-Ramil, J. Higman, H.C. Sharp, N. Smith, An empirical study of the evolution of an agile-developed software system, in: 29th International Conference on Software Engineering, 2007, Minneapolis, MN, pp. 511–518.
[39] S.J. Carriere, R. Kazman, S.G. Woods, Assessing and maintaining architectural quality, in: Proceedings of the Third European Conference on Software Maintenance and Reengineering, Amsterdam, 1999, pp. 22–30.
[40] N. Chapin, J.E. Hale, K.M. Khan, J.F. Ramil, W.-G. Tan, Types of software evolution, software maintenance, Journal of Software Maintenance. Research, Practice 13 (1) (2001) 3–30.
[41] M.A. Chaumun, H. Kabaili, R.K. Keller, F. Lustman, A change impact model for changeability assessment in object-oriented software systems, in: Proceedings of the Third European Conference on Software Maintenance and Reengineering, 1999, pp. 130–138.
[42] M.A. Chaumun, H. Kabaili, R.K. Keller, F. Lustman, A change impact model for changeability assessment in object-oriented software systems, Science of Computer Programming 45 (2) (2002) 155–177.
[43] O.C. Chesley, X. Ren, B.G. Ryder, Crisp: a debugging tool for Java programs, in: Proceedings of the International Conference on Sotware Maintenance, 2005, pp. 401–410.
[44] P. Clarke, B. Malloy, P. Gibson, Using a taxonomy tool to identify changes in Oo software, in: Proceedings of the Seventh European Conference on Software Maintenance and Reengineering, 2003, pp. 213–222.
[45] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, J. Stafford, Documenting Software Architectures: Views and Beyond, Addison-Wesley, 2003.
[46] R. Conradi, M.N. Nguyen, A.I. Wang, Planning support to software process evolution, International Journal of Software Engineering, Knowledge Engineering 10 (1) (2000) 31–47.
[47] S. Cook, R. Harrison, P. Wernick, A simulation model of self-organising evolvability in software systems, in: IEEE International Workshop on Software Evolvability, 2005, pp. 17–22.
[48] R.T. Crocker, A. Von Mayrhauser, Maintenance support needs for object-oriented software, in: Proceedings of the Seventeenth Annual International Computer Software and Applications Conference, Phoenix, AZ, 1993, pp. 63–69.
[49] S. Dekleva, Delphi study of software maintenance problems, in: Proceerdings of the Conference on Software Maintenance, Orlando, FL, 1992, pp. 10–17.
[50] S.G. Eick, T.L. Graves, A.F. Karr, J.S. Marron, A. Mockus, Does code decay? Assessing the evidence from change management data, IEEE Transactions on Software Engineering 27 (1) (2001) 1–12.
[51] S.G. Eick, T.L. Graves, A.F. Karr, A. Mockus, P. Schuster, Visualizing software changes, IEEE Transactions on Software Engineering 28 (4) (2002) 396–412.
[52] T. Feng, J.I. Maletic, Applying dynamic change impact analysis in component-based architecture design, in: Seventh ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2006, pp. 43–48.
[53] R. Fiutem, P. Tonella, G. Antoniol, E. Merlo, A cliche-based environment to support architectural reverse engineering, in: Proceedings of the Third Working Conference on Reverse Engineering, 1996, pp. 319–328.
[54] B. Fluri, H.C. Gall, Classifying change types for qualifying change couplings, in: Proceedings of the 14th IEEE Conference on Program Comprehension, Athens, Greece, 2006, pp. 35–45.
[55] M. Fowler, Refactoring: Improving the Design of Existing Code, 2000, Addison-Wesley, Upper Saddle River, MJ.
[56] M. Fredericks, V. Basili, Using Defect Tracking and Analysis to Improve Software Quality, SP0700-98-4000, DOD Data & Analysis Center for Software (DACS), Rome, NY, 1998.
[57] H. Gall, M. Jazayeri, R.R. Klosch, G. Trausmuth, Software evolution observations based on product release history, in: Proceedings of the International Conference on Software Maintenance, 1997, pp. 160–166.
[58] J. Grundy, J. Hosking, High-level static and dynamic visualisation of software architectures, in: Proceedings of the 2000 IEEE International Symposium on Visual Languages, Seattle, WA, 2000, pp. 5–12.
[59] J. Gustafsson, J. Paakki, L. Nenonen, A.I. Verkamo, Architecture-centric software evolution by software metrics and design patterns, in: Proceedings fo the Sixth European Conference on Software Maintenance and Reengineering, Budapest, 2002, pp. 108–115.
[60] A. Hac, Software renovation for large software applications, in: IEEE Region 10 International Conference (TENCON '92), Melbourne, Australia, 1992, pp. 307–311.
[61] J.E. Hannay, D.I.K. Sjoberg, T. Dyba, A systematic review of theory use in software engineering experiments, IEEE Transactions on Software Engineering 33 (2) (2007) 87–107.
[62] G.T. Heineman, A. Mehtra, Architectural evolution of legacy systems, in: Proceedings of the 23rd Annual International Computer Software and Applications Conference, Phoenix, AZ, 1999, pp. 4–12.
[63] I. Herraiz, G. Robles, J.M. Gonzalez-Barahona, A. Capiluppi, J.F. Ramil, Comparison between slocs and number of files as size metrics for software evolution analysis, in: Proceedings of the 10th European Conference on Software Maintenance and Reengineering, 2006, pp. 8.
[64] D.S. Hinley, Software evolution management: a process-oriented perspective, Information and Software Technology 38 (11) (1996) 723–730.
[65] L. Hochstein, M. Lindvall, Combating architectural degeneration: a survey, Information and Software Technology 47 (10) (2005) 643–656.
[66] P. Hsia, A. Gupta, C. Kung, J. Peng, S. Liu, A study on the effect of architecture on maintainability of object-oriented systems, in: Proceedings of the International Conference on Software Maintenance, 1995, pp. 4–11.
[67] I. Ivkovic, K. Kontogiannis, A framework for software architecture refactoring using model transformations and semantic annotations, in: Proceedings of the Conference on Software Maintenance and Reengineering, 2006, pp. 135–144.
[68] M. Jorgensen, M. Shepperd, A systematic review of software development cost estimation studies, IEEE Transactions on Software Engineering 33 (1) (2007) 33–53.
[69] C.F. Kemerer, S. Slaughter, An empirical approach to studying software evolution, IEEE Transactions on Software Engineering 25 (4) (1999) 493–503.
[70] S. Kim, E.J. Whitehead, J. Bevan, Analysis of Signature Change Patterns, in: Proceedings of the 2005 International Workshop on Mining Software Repositories, ACM Press, St. Louis, MO, 2005, pp. 1–5.
[71] S. Kim, E.J. Whitehead, Y. Zhang, Classifying software changes: clean or buggy?, IEEE Transactions on Software Engineering 34 (2) (2008) 181–196
[72] B. Kitchenham, Procedures for performing systematic reviews, TR/SE-0401, Department of Computer Science, Keele University, 2004, Keele, Staffs, UK.
[73] B. Kitchenham, E. Mendes, G. Travassos, Cross versus within-company cost estimation studies: a systematic review, IEEE Transactions on Software Engineering 33 (5) (2007) 316–329.
[74] B.A. Kitchenham, G.H. Travassos, A.V. Mayrhauser, F. Niessink, N.F. Schneidewind, J. Singer, S. Takada, R. Vehvilainen, H. Yang, Towards an Ontology of Software Maintenance, Journal of Software Maintenance: Research and Practice 11 (6) (1999) 365–389.
[75] P. Kruchten, Architectural blueprints—the "4+ 1" view model of software architecture, IEEE Software 12 (6) (1995) 42–50.
[76] D. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, C. Chen, Change impact identification in object oriented software maintenance, in: Proceedings of the International Conference on Software Maintenance, Victoria, BC, 1994, pp. 202–211.

[77] W. Lam, V. Shankararaman, Managing change in software development using a process improvement approach, in: Proceedings of the 24th Annual Euromicro Conference, Vasteras, 1998, pp. 779–786.
[78] W. Lam, V. Shankararaman, Requirements change: a dissection of management issues, in: Proceedings of the 25th EuroMicro Conference, 1999, Milan, Italy, pp. 244–251.
[79] N. Lassing, D. Rijsenbrij, H. Van Vliet, Flexibility of the combad architecture," in: Proceedings of the First Working IFIP Conference on Software Architecture, 1999, pp. 357–368.
[80] N. Lassing, D. Rijsenbrij, H. Van Vliet, Towards a broader view on software architecture analysis of flexibility, in: Proceedings of the Sixth Asia Pacific Conference on Software Engineering, 1999, pp. 238–245.
[81] Y. Lee, J. Yang, K.H. Chang, Metrics and evolution in open source software, in: Seventh International Conference on Quality Software, 2007, pp. 191–197.
[82] M.M. Lehman, Programs, life cycles and laws of software evolution, Proceedings of the IEEE 68 (9) (1980) 1060–1076.
[83] M.M. Lehman, Feedback, evolution and software technology, in: Proceedings of the 10th International Process Support of Software Product Lines Software Process Workshop, 1996, pp. 101–103.
[84] M.M. Lehman, L. Belady, Software Evolution – Processes of Software Change, Academic Press, London, 1985.
[85] M.M. Lehman, D.E. Perry, J.F. Ramil, Implications of evolution metrics on software maintenance, in: Proceedings of the International Conference on Software Maintenance, Bethesda, MD, 1998, pp. 208–217.
[86] M.M. Lehman, D.E. Perry, J.F. Ramil, On evidence supporting the feast hypothesis and the laws of software evolution, in: Proceedings of the Fifth International Software Metrics Symposium, 1998, pp. 84–88.
[87] M.M. Lehman, J.F. Ramil, Towards a theory of software evolution – and its practical impact, in: Proceedings of the International Symposium on Principles of Software Evolution, 2000, pp. 2–11.
[88] M.M. Lehman, J.F. Ramil, P.D. Wernick, D.E. Perry, W.M. Turski, Metrics and laws of software evolution-the nineties View, in: Proceedings of the Fourth International Software Metrics Symposium, 1997, pp. 20–32.
[89] J.J. Li, D.B. Mulcare, W.E. Wong, Dependability of complex software systems with component upgrading, in: Computer Software and Applications Conference, 2000. COMPSAC 2000 The 24th Annual International, 2000, pp. 413–418.
[90] B. Lientz, B. Swanson, Software maintenance management, Addison-Wesley, 1980.
[91] I.H. Lin, D.A. Gustafson, Classifying software maintenance, in: Proceedings of the Conference on Software Maintenance, Scottsdale, AZ, 1988, pp. 241–247.
[92] M. Lindvall, K. Sandahl, How well do experienced software developers predict software change?, Journal of Systems and Software 43 (1) (1998) 19–27.
[93] M. Lindvall, R. Tesoriero, P. Costa, Avoiding architectural degeneration: an evaluation process for software architecture, in: Proceedings of the Eighth IEEE Symposium on Software Metrics, 2002, pp. 77–86.
[94] N.H. Madhavji, The prism model of changes, in: Proceedings of the 13th International Conference on Software Engineering, Austin, TX, 1991, pp. 166–177.
[95] N.H. Madhavji, Environment evolution: The prism model of changes, IEEE Transactions on Software Engineering 18 (5) (1992) 380–392.
[96] J.F. Maranzano, S.A. Rozsypal, G.H. Zimmerman, G.W. Warnken, P.E. Wirth, D.M. Weiss, Architecture reviews: experience and practice, IEEE Software 22 (2) (2005) 34–43.
[97] P.J. Mayhew, C.J. Worsley, P.A. Dearnley, Control of software prototyping process: change classification approach, Information and Software Technology 31 (2) (1989) 59–66.
[98] D.S. Mccrickard, G.D. Abowd, Assessing the impact of changes at the architectural level: a case study on graphical debuggers, in: Proceedings of the International Conference on Software Maintenance, Monterey, CA, 1996, pp. 59–67.
[99] E. Mendes, A systematic review of web engineering research, in: International Symposium on Empirical Software Engineering, 2005, pp. 498–507.
[100] T. Mens, J. Buckley, M. Zenger, A. Rashid, Towards a taxonomy of software evolution, in: Proceedings of the International Workshop on Unanticipated Software Evolution, Warsaw, Poland, 2003, pp. 1–18.
[101] T. Mens, T. Tourwe, A survey of software refactoring, IEEE Transactions on Software Engineering 30 (2) (2004) 126–139.
[102] R. Mittermeir, Software evolution: let's sharpen the terminology before sharpening (out-of-scope) tools, in: Proceedings of the 4th International Workshop on Principles of Software Evolution, ACM Press, Vienna, Austria, 2001, pp. 114–121.
[103] A. Mockus, S.G. Eick, T. Graves, A. Karr, On Measurement and Analysis of Software Changes", Doc. No. ITD-99-36760F, BL0113590-990401-06TM, National Institute of Statistical Sciences, 1999.
[104] A. Mockus, L.G. Votta, Identifying reasons for software changes using historic databases, in: Proceedings of the International Conference on Software Maintenance, San Jose, CA, 2000, pp. 120–130.
[105] A. Mockus, D. Weiss, Predicting risk of software changes, Bell Labs Technical Journal 5 (2) (2000) 169–180.
[106] P. Mohagheghi, R. Conradi, An empirical study of software change: origin, acceptance rate, and functionality vs. quality attributes, in: Proceedings of the 2004 International Symposium on Empirical Software Engineering (ISESE '04), 2004, pp. 7–16.
[107] V. Nanda, N.H. Madhavji, The impact of environmental evolution on requirements changes, in: Proceedings of the International Conference on Software Maintenance, 2002, pp. 452–461.
[108] J. Nedstam, E.A. Karlsson, M. Host, The architectural change process, in: Proceedings of the 2004 International Symposium on Empirical Software Engineering (ISESE '04), 2004, pp. 27–36.
[109] M.N. Nguyen, R. Conradi, Classification of meta-processes and their models, in: Proceedings of the Third International Conference on the Software Process, 1994, pp. 167–175.
[110] N. Nguyen, Framework and Approach for Managing Software Process Evolution in Epos, NTNU, Trondheim, Norway, Doctoral Thesis, 1997.
[111] N. Nurmuliani, D. Zowghi, S.P. Williams, Using card sorting technique to classify requirements change, in: Proceedings of the 12th IEEE International Requirements Engineering Conference, 2004, pp. 240–248.
[112] J.S. O'neal, D.L. Carver, Analyzing the impact of changing requirements, in: Proceedings of the International Conference on Software Maintenance, Florence, 2001, pp. 190–195.
[113] C. O'reilly, P. Morrow, D. Bustard, Lightweight prevention of architectural erosion, in: Proceedings of the Sixth International Workship on Principles of Software Evolution, Helsinki, Finland, 2003, pp. 59–64.
[114] D.L. Parnas, Software Aging, in: Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy, 1994¸ pp. 279–287.
[115] N. Phillips, S. Black, Distinguishing between learning, growth and evolution, in: IEEE International Workshop on Software Evolvability, 2005, pp. 49–52.
[116] D. Port, H. Liguo, Strategic architectural flexibility, in: Proceedings of the International Conference on Software Maintenance, Amsterdam, The Netherlands, 2003, pp. 389–396.
[117] A. Postma, P. America, J.G. Wijnstra, Component replacement in a long-living architecture: the 3rdba approach, in: Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA '04), 2004, pp. 89–98.
[118] J.S. Poulin, Evolution of a software architecture for management information systems, in: Joint Proceedings of the Second International Software Architecture Workshop and International Workshop on Multiple Perspectives in Software Development, ACM Press, San Francisco, CA, 1996, pp. 134–137.
[119] R. Purushothaman, D.E. Perry, Toward understanding the rhetoric of small source code changes, IEEE Transactions on Software Engineering 31 (6) (2005) 511–526.
[120] J.F. Ramil, Laws of software evolution and their empirical support, in: Proceedings of the International Conference on Software Maintenance, 2002, pp. 71–71.
[121] J. Ratzinger, M. Fischer, H. Gall, Improving evolvability through refactoring, in: Proceedings of the 2005 International Workshop on Mining Software Repositories, ACM Press, St. Louis, MO, 2005, pp. 1–5.
[122] X. Ren, F. Shah, F. Tip, B.G. Ryder, O. Chesley, Chianti: a tool for change impact analysis of java programs, in: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, Springer, Vancouver, BC, Canada, 2004, pp. 432–448.
[123] J.E. Robbins, D.M. Hilbert, D.F. Redmiles, Using critics to analyze evolving architectures, in: Joint Proceedings of the Second International Software Architecture Workshop and International Workshop on Multiple Perspectives in Software Development, ACM Press, San Francisco, CA, 1996, pp. 90–93.
[124] H. Rombach, B. Ulery, J. Valett, Toward Full Life Cycle Control: Adding Maintenance Measurement to the Sel, Journal of Systems and Software 18 (2) (1992) 125–135.
[125] N. Sadou, D. Tamzalit, M. Oussalah, A unified approach for software architecture evolution at different abstraction levels, in: Proceedings of the 2005 Eighth International Workshop on Principles of Software Evolution, 2005, pp. 65–68.
[126] S.R. Schach, B. Jin, L. Yu, G.Z. Heller, J. Offutt, Determining the Distribution of Maintenance Categories: Survey Versus Measurement, Empirical Software Engineering 8 (4) (2003) 351–365.
[127] M. Shaw, D. Garlan, Software Architecture: Perspectives on an Emerging Discipline, Prentice Hall, Upper Saddle River, NJ, 1996.
[128] I. Sommerville, Software Engineering, 7th ed., Addison-Wesley, 2004.
[129] M. Stoerzer, B.G. Ryder, X. Ren, F. Tip, Change Classification and Its Applications to Program Development and Testing, Technical Report DCS-TR-05-566, Department of Computer Science, Rutgers University, 2005.
[130] R.C. Sugden, M.R. Strens, Strategies, tactics and methods for handling change, in: Proceedings of the IEEE Symposium and Workshop on Engineering of Computer-Based Systems, Friedrichshafen, 1996, pp. 457–463.
[131] M. Svahnberg, C. Wohlin, An investigation of a method for identifying a software architecture candidate with respect to quality attributes, Empirical Software Engineering 10 (2) (2005) 149–181.
[132] M. Svahnberg, C. Wohlin, L. Lundberg, M. Mattsson, Quality attribute-driven selection of software architecture structures, in: Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering, ACM Press, Ischia, Italy, 2002, pp. 819–826.
[133] B. Swanson, The dimensions of maintenance, in: Proceedings of the 2nd International Conference on Software Engineering, IEEE Computer Society Press, San Francisco, CA, 1976, pp. 492–497.

[134] L. Tahvildari, R. Gregory, K. Kontogiannis, An approach for measuring software evolution using source code features, in: Proceedings of the Sixth Asia Pacific Software Engineering Conference, Takamatsu, 1999, pp. 10–17.
[135] J.B. Tran, R.C. Holt, Forward and reverse repair of software architecture, in: Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research, IBM Press, Mississauga, Ontario, Canada, 1999, pp. 12–21.
[136] Q. Tu, M.W. Godfrey, An integrated approach for studying architectural evolution, in: Proceedings of the 10th International Workshop on Program Comprehension, 2002, pp. 127–136.
[137] R.T. Tvedt, P. Costa, M. Lindvall, Does the code match the design? A process for architecture evaluation, in: Proceedings of the International Conference on Software Maintenance, 2002, pp. 393–401.
[138] R.T. Tvedt, M. Lindvall, P. Costa, A process for software architecture evaluation using metrics, in: Proceedings of the 27th Annual NASA Goddard/IEEE Software Engineering Workshop, 2002, pp. 191–196.
[139] F. Van Rysselberghe, S. Demeyer, Mining version control systems for Facs (frequently applied changes), in: 26th International Conference on Software Engineering, Edinburgh, Scotland, 2004, pp. 48–52.
[140] P. Weissgerber, S. Diehl, Identifying refactorings from source-code changes, in: 21st IEEE/ACM International Conference on Automated Software Engineering, 2006, pp. 231–240.
[141] Z. Xing, E. Stroulia, Analyzing the evolutionary history of the logical design of object-oriented software, IEEE Transactions on Software Engineering 31 (10) (2005) 850–868.
[142] Z. Xing, E. Stroulia, Umldiff: an algorithm for object-oriented design differencing, in: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, ACM Press, Long Beach, CA, 2005.
[143] Y. Yang, C. Riva, Scenarios for mining the software architecture evolution, in: Proceedings of the 2006 International Workshop on Mining Software Repositories, ACM Press, Shanghai, China, 2006, pp. 10–13.