Monitoring Software Quality Evolution by Analyzing Deviation Trends of Modularity Views

Tianmei Zhu[1], Yijian Wu[1], Xin Peng[1], Zhenchang Xing[2], Wenyun Zhao[1]
[1]School of Computer Science, Fudan University, Shanghai, China
[2]School of Computing, National University of Singapore, Singapore
{09212010034, wuyijian, pengxin, wyzhao}@fudan.edu.cn
xingzc@comp.nus.edu.sg

Abstract—**In the long-term evolution of software systems, various maintenance activities such as functionality extension, bug fixing, refactoring may positively or negatively affect the quality of design and implementation. The trend of quality degradation caused by negative affections may accumulate and cause serious difficulties for future maintenance of the software if they were not addressed properly in time. In this paper, we propose an approach for monitoring the degradation trends of software design in evolution and providing useful feedbacks for evolution decisions. The approach is based on the assumption that the deviations between different modularity views and their trends in evolution can be used to monitor the degradation trends of design. Currently, our approach considers three modularity views, namely package view, structural cluster view and semantic cluster view. Package view denotes the package structure reflecting the desired modularity view; Structural cluster view and semantic cluster view are the modularity views extracted from implementation by software clustering based on formal information and non-formal information, respectively. Then based on the three modularity views extracted from each version, our approach calculates the similarity between different views as the measurement of modularity deviations, and analyzes the deviation trends over a series of versions. We conduct an empirical study on three open-source systems, which confirms that continuous monitoring of deviation trends of modularity views can provide useful feedbacks for future evolution decisions.**

Keywords- evolution analysis, software quality evolution, software modulairty, software clustering, maintenance history

I. Introduction

Software usually undergoes a continuous process of evolution driven by evolutionary development processes, design improvement, bug fixing, and request for new features [1], [2], [9]. In the long-term evolution, maintenance actions are made for specific evolution intent and objectives in different phases [2]. For example, in some phases a large number of new features can be introduced, and in other phases the design structures can be refactored to provide better maintainability and extensibility for future evolution. These maintenance actions may positively or negatively affect the quality of design and implementation of a software system and result in the deviation of the modularity quality from the desired level. The trend of quality degradation caused by negative affections may accumulate and cause serious difficulties for future maintenance of the software if they were not addressed properly in time. Therefore, monitoring and controlling the trends of software quality evolution is essential for high-efficiency software maintenance.

Modularity is extremely important for software development and evolution. Good modularity can improve the flexibility and comprehensibility of the software system [3], [4], while bad modularity can cause expensive refactoring and software defects [5]. Thus, modularity is often used as an important criterion for evaluating the quality of software design and implementation [39].

For software systems developed in object-oriented (OO) languages such as Java, the packages (or namespaces used) largely reflect the desired modularity view of the designers. This modularity view may be carefully designed initially. However, as software evolves, it may be violated frequently. For example, newly introduced elements may be put into an inappropriate existing module or may result in inappropriate inter-module interactions. This may violate well-defined modularity principles such as information hiding and functional independence.

Software clustering techniques [6], [7] are often used to recover the modularity views implied in the actual implementation. More specifically, software clustering can be divided into structural clustering and semantic clustering, depending on the information used for clustering. Structural clustering is based on formal information such as method calls and variable references, while semantic clustering is based on non-formal information such as identifiers and comments [6].

Using structural clustering and semantic clustering, we can recover two different kinds of implied modularity views from the implementation. Structural clusters group program units by structural relations implemented in source code thus can be considered as a modularity view reflecting the *de-facto* implementation. Semantic clusters are groups of program units that use similar vocabulary and reveal linguistic topics and intention of the code [7], and thus can be regarded as providing a modularity view by the implemented concerns and intentions.

These two modularity views recovered by software clustering, together with the package structure (the desired modularity view), constitute the three modularity views discussed in this paper. In an ideal modular design and implementation, the three modularity views are well aligned. For example, implementation units for relevant concerns are within the same package, which presents good structural characteristics of high cohesion and low coupling. Therefore, the alignment among the three modularity views may indicate the quality of modular design and implementation.

We hypothesize that we can monitor software quality evolution and provide useful feedback for evolution decisions by analyzing the deviation trends of the three modularity views. We are only interested in the quality essential in software evolution, such as maintainability. To validate our hypothesis, we propose an approach for monitoring quality evolution based on the alignment among three preceding modularity views and conduct an empirical study on three open-source systems.

Our approach involves three main activities. First, we extract the three modularity views from each version of the subject systems using automatic program analysis and clustering techniques [24]. Second, we calculate similarity or alignment measurements among the three modularity views. Third, we perform a longitudinal analysis of the deviation trends over a series of the versions of the subject systems.

Our empirical study focuses on the following research questions: 1) What deviation trends of different modularity views does a software system show during its evolution? 2) How do we understand software quality evolution by analyzing the deviation trends? 3) Can we get useful feedback for evolution decisions by monitoring the deviation trends of modularity views?

Our contribution in this paper is three-fold. First, we propose an approach for quality evolution monitoring based on the analysis of deviation trends of different modularity views. Second, we present an empirical study that confirms that continuous monitoring of deviation trends of modularity views can provide useful feedbacks for future evolution decisions. Third, we identify some typical deviation trend patterns of modularity views that are proved to be useful for reflecting quality evolution.

The remainder of this paper is organized as follows. Section II summarizes related work on software evolution analysis and software modularity. Section III defines basic terminology used in our study. Section IV introduces our approach for monitoring deviation trends of different modularity views. Section V presents our empirical study on three open source Java systems and explains the result with concrete examples from our study. Section VI discusses threats to validity of our approach and experiments. Section VII concludes our work and findings and outlines future work.

II. Related Work

A. Monitoring and understanding software evolution

Evolution is an essential characteristic of software systems. There have been several research efforts to date aiming at understanding evolution in open source software. In [45], Breivold *et al.* made a systematic review of studies of open source software evolution. Godfrey and Qiang took a case study about open source software projects' evolution in [41]. Meanwhile, another line of research has focused on the investigation of how metrics can be applied to software evolution [43], [44]. In [8], Mens and Demeyer provided a classification of various approaches that use metrics to understand, analyze, control and improve the software evolution process, namely predictive analysis and retrospective analysis. Software growth metrics such as line of source code (LOC) and number of modules are usually used to monitor the software evolution [41], [9], [10].

Exploiting the development history is a widely used approach. Ali and Maqbool proposed an approach to monitor software evolution using multiple types of changes [42]. Mockus and Votta showed that textual descriptions of software change are useful in version control system and proposed suggestions to utilize change data to diagnose the state of a software project [13]. This implies that software quality evolution can be monitored and inferred by analyzing software maintenances actions. Van Rysselberghe and Demeyer proposed an approach to find unstable components, coherent entities and design changes by visualizing change history [1].

Xing and Stroulia presented a method for understanding software evolution by analyzing design-level structural changes in source code [12]. The changes detected by a differencing tool were categorized into several types to show whether the development is under rapid expansion or just steadily going. This indicates various evolution phases for maintainers to plan necessary actions to keep maintainability of the software. Instead of comparing code changes, our approach analyzes modularity deviations to find the change trend for evolution.

B. Software modularity

There are several ways to describe software modularity. Different expressions on software modularity reflect different design considerations. Cohesion and coupling metrics [14] are frequently used to measure *absolute* values of the degree to which modules are encapsulated, or interrelated with each other. Structural [16] and conceptual [17], [18], [19] information is considered in measuring coupling and cohesion, providing different modularity views. However, what is the relationship between these two modularity views has not been thoroughly investigated. The deviations of different modularity views express a *relative* measure that shows how a system is consistent in structure as inspected from various perspectives.

There have been researches in checking the consistency of modularity views between design and implementation [3], [5]. However, we only consider design modularity, and analyze the deviations between different views. We assume that when viewed from different perspectives, the modularity views may be in nature different and the deviations between them are an indicator for software quality problems.

A commonly used way to obtain implied modularity is software clustering [15], which exploits implicit relationships among modules. It is usually applied for understanding the inherent structure of complex systems [7], [20], [21] or for discovering a better design of systems [22], [23]. Cluster analysis has been proved useful for re-modularizing legacy systems [24], and various algorithms have been used for various software analysis purposes [6], [20], [22], [23].

Recently, clustering algorithms have been also applied in analyzing software evolution. Wu *et al.* applied several clustering algorithms on the history of five open source systems to find how well each algorithm with different parameters performs [25]. Kothari *et al.* focused on change clusters extracted from the evolution history of a software system to help project managers to classify different code change activities and monitor the progress of the project [26]. These approaches either focus on evaluation clustering algorithm or use software changes as input to the clustering algorithm. On the contrary, we apply software clustering techniques to different aspects of the same version of software to obtain two different kinds of modularity views of that version, and then we compare and contrast these modularity views and desired modularity views to investigate the deviation trends of software quality.

III. CONCEPTS

In this section, we explain a set of important concepts in our approach, such as modularity views and deviation trends.

A. Modularity views

A modularity view is a modular structure of the target system that is desired by designers or implied in the implementation. Specifically, in this paper, we consider three modularity views, namely Package View, Structural Cluster View and Semantic Cluster View.

1) Package View (Pkg)

Package view prescribes how developers intentionally group related source files as modules. It is the desired modularity view of the designers.

2) Structural Cluster View (St)

Structural cluster view is the modularity view implied by file-level based structural clusters. It reflects the nature of inter-file dependencies and method invocation relations.

3) Semantic Cluster View (Se)

Semantic cluster view presents the semantically cohesive conceptual structure implied by file-level based semantic clusters. It reflects the nature of vocabulary used and topics involved in different source files and their correlations.

B. Modularity Deviations

An essential assumption of our work is that the deviation trends between different modularity views can be used as indicators for quality evolution monitoring. If different modularity views of a system are well aligned, the developers can easily locate relevant concepts and localize the changes in single packages. Otherwise, the developers may need to explore a series of places scattered in different packages for specific tasks of maintenance, which is time-consuming and error-prone.

A well-designed package structure groups semantically and structurally relevant design elements into the same package. It is usually the case that the desired design prescribed by the package view is well followed initially. However, as software evolves, especially due to quick and dirty changes, deviations between the desired modularity view and the implied modularity views implied by the implementation may be introduced and accumulated. The developers usually can reduce the deviations by reorganizing the packages to make them better aligned with the implied modularity views.

C. ***Si**milarity between the **Mo**dularity views (SiMo)*

To measure the deviations between two different modularity views, we propose to use a set of *SiMo* (Similarity between the Modularity views) metrics. The smaller the *SiMo* metrics are, the higher the deviations are. We use the following three *SiMo* metrics in our approach:

S_{Pkg_St}, i.e., the similarity between package view and structural cluster view;

S_{Pkg_Se}, i.e., the similarity between package view and semantic cluster view;

S_{St_Se}, i.e., the similarity between structural cluster view and semantic cluster view.

We apply MoJoFM method [27] to quantify the three *SiMo* metrics, which are explained in Section IV.

D. Deviation Trends as Indicators of Quality Evolution

After we compute the modularity deviations of individual versions, we perform longitudinal analysis to analyze the deviation trends in a sequence of versions as the indicators for quality evolution. Given a *SiMo* metric, a similarity change is classified as one of the three following types: a remarkable increase of similarity is denoted as a *rise* ("$\nearrow$"); a remarkable decrease of similarity is denoted as a *drop* ("$\searrow$"); otherwise, no significant change of similarity is found, denoted as a *hold* ("$\rightarrow$"). The criteria of classifying a change to be a *rise*, *drop* or *hold* is project-specific and experiential, which is discussed in Section IV.

Based on the concept of similarity changes, we define the deviation trend of modularity views as a combination of the changes of the three *SiMo* metrics. Formally, a deviation trend is a three-tuple (*Similarity Change of* S_{Pkg_St}, *Similarity Change of* S_{Pkg_Se}, *Similarity Change of* S_{St_Se}). For example, for an evolution that involves the changes of S_{Pkg_St} *rise*, S_{Pkg_Se} *drop*, and S_{St_Se} *hold*, the deviation trend can be denoted as ($\nearrow$, $\searrow$, $\rightarrow$).

IV. METHODOLOGY

A. Overview

Figure 1 presents an overview of our approach and our empirical study. The three main activities involved include construction of modularity views, computation of similarity metrics and analysis of deviation trends.

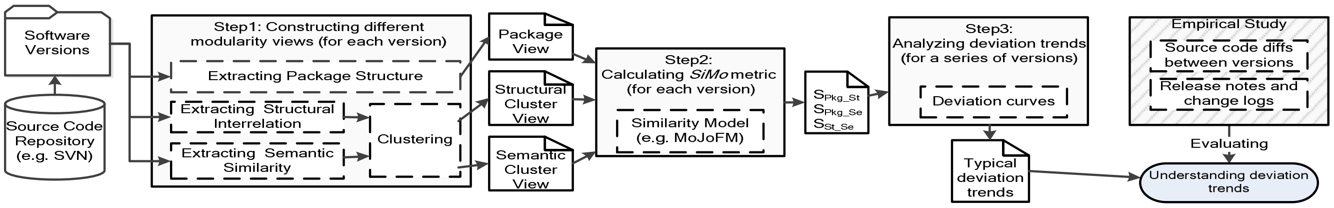
Software Versions
Source Code Repository (e.g. SVN)
Step1: Constructing different modularity views (for each version)
Extracting Package Structure
Extracting Structural Interrelation
Extracting Semantic Similarity
Clustering
Package View
Structural Cluster View
Semantic Cluster View
Step2: Calculating SiMo metric (for each version)
Similarity Model (e.g. MoJoFM)
SPkg_St
SPkg_Se
SSt_Se
Step3: Analyzing deviation trends (for a series of versions)
Deviation curves
Typical deviation trends
Empirical Study
Source code diffs between versions
Release notes and change logs
Evaluating
Understanding deviation trends

Figure 1. An overview of the monitoring process and the empirical study

Based on the source code of each version of a software system, we first construct three modularity views by static code analysis and software clustering. Then, we compute the three *SiMo* metrics between different modularity views. Finally, we evaluate the deviation trends of the three *SiMo* metrics in a sequence of consecutive versions.

To evaluate the effectiveness of our approach and answer the research questions raised in Section I, we conducted an empirical study in which we investigated the evolution history of three open-source systems. In the empirical study, we identified typical patterns of modularity deviation trends. Furthermore, we related the patterns of deviation trends with the evolution intention, using several information sources such as release notes, change logs, and design differences.

B. Extraction of Modularity Views

In our approach, we extract the following three modularity views of each individual version of the software system.

1) Package view

Package view can be directly extracted from the package structure implemented in each version of the software system. Since the similarity comparison algorithm MoJoFM works only on flat decompositions, we use the leaf-level package partition ignoring the package hierarchy so that the package view can be compared with the other two modularity views obtained by clustering.

2) Structural cluster view

In this study, we consider the following types of structural dependencies for structural clustering, including inheritance, interface implementation, aggregation, and usage dependency (e.g., method calls, method parameters, local variables). And relationships among classes are "lifted" to file level. We use the Design Structure Matrix [40] to describe the structural dependencies among source files. Each row or column represents a source file. Each cell is 1 if the two corresponding files are related, or 0 otherwise. In this representation, each source file is represented as a vector that describes how it is related to the other files. The similarity between source files is measured by computing the cosine similarity between two vectors of the two files.

Based on the structural similarities between source files, we apply agglomerative hierarchical algorithm (implemented in Weka [29]) to produce the structural modularity view. The agglomerative algorithm starts with an initial cluster set that treats each file as a cluster. After that, an iterative process is conducted to merge two clusters with the largest similarity iteratively until all the clusters are merged into one cluster. To get a partition of the systems, we would use a cut-point height [6] to output the clusters at specific level as the results of clustering. In our current implementation, the cut point height is automatically determined in that the number of clusters produced is as close as possible to the number of packages in the package view.

3) Semantic cluster view

In order to extract semantic information of the subject system, we preprocess the source code files to eliminate non-textual tokens (i.e., operators, numbers, etc.), Java keywords and stop words and to split identifiers (e.g. based on case switching and underscore). Then, we build a term-document matrix. Next, we use Latent Semantic Indexing (LSI) [28] to compute the concept space of the subject system. LSI is an indexing and retrieval technique that uses a mathematical technique called Singular Value Decomposition (SVD) to identify patterns in the relationships between the terms and concepts contained in an unstructured collection of text. SVD reduces the number of dimensions and represents documents as vectors in the reduced concept space. In this paper, we take similar choice as [22] to determine the dimensions of the concept space, that is, the number of concepts in concept space is the number of singular values in diagonal matrix S greater than 1. Finally, we get the corresponding vectors of the source files in the concept space.

The similarity measurement and clustering algorithm used in semantic clustering are the same as those used in the structural clustering.

C. SiMo Metrics Computation

In our approach, the three *SiMo* metrics are calculated using a widely adopted similarity comparison method, MoJoFM [27]. MoJoFM is a normalized revision of MoJo [30], [31] that calculates the minimum number of Move-and-Join operations to transform one partition into another. Given two partitions A and B, the similarity between A and B can be measured by MoJoFM with the formula

$$\mathrm{MoJoFM(A,B)} = \left(1 - \frac{\mathrm{mno(A,B)}}{\max\big(\mathrm{mno}(\forall\mathrm{A},\mathrm{B})\big)}\right) \times 100\%$$

where mno(A,B) means the minimum number of Move-and-Join operations to transform partition A to partition B, $\max\big(\mathrm{mno}(\forall\mathrm{A},\mathrm{B})\big)$ means the maximum distance from any partitions to partition B (see [27] for details). MoJoFM ranges from 0% to 100%. The larger the MoJoFM value is, the more similar the two partitions are.

The three *SiMo* metrics can then be defined as:

S_{Pkg_St}=MoJoFM(*Pkg*, *St*),

S_{Pkg_Se}=MoJoFM(*Pkg*, *Se*), and

S_{St_Se}=Max(MoJoFM(*St*, *Se*), MoJoFM(*Se*, *St*)).

Note that S_{Pkg_St} and S_{Pkg_Se} are asymmetric. The reason is that deciding how to organize a package is subjective.

Designers may take a structural biased consideration or a semantic biased consideration to organize the packages. Therefore, we simply measure to what degree the package view deviates from cluster views, ignoring the reverse deviation.

D. Analyzing deviation trends

Based on the *SiMo* metrics between different modularity views, three curves of similarity metrics are obtained for a software system (see Figure 2, 3, 4 for examples). Change trends of each *SiMo* metric can be defined in a project-specific and experiential manner. In our study, a *hold* is defined as the change of similarity metric being within 1 percentage of the MoJoFM value (approximations are used instead of the real difference). Any changes beyond 1 percentage range are regarded as *rise*s or *drop*s respectively.

As mentioned in Section II, a deviation trend of different modularity views is a combination of three change trends (i.e. *rise*, *drop*, *hold*) of three *SiMo* metrics (i.e. S_{Pkg_St}, S_{Pkg_Se}, S_{St_Se}).There can be up to 3^3=27 patterns of deviation trends. We expect to encounter some of these patterns in software evolution history, indicating the issues of software quality evolution, in our follow-up empirical study.

V. Empirical Study

A. Research questions and subject systems

In our empirical study, we aim at answering the following research questions:

Q1. What deviation trends of different modularity views does a software system show during its evolution?

Q2. How do we understand software quality evolution by analyzing the deviation trends?

Q3. Can we get useful feedback for evolution decisions by monitoring the deviation trends of modularity views?

To answer these questions, we conducted an empirical study on three open-source Java systems, JFreeChart [32], JHotDraw [33], [34] and JEdit [35]. We picked these three subject systems not only because they are well-known and believed to be well maintained, but also because they have rich sets of release notes and documents to confirm our observations.

All three subject systems have been developed and maintained for about ten years. The projects JEdit and JHotDraw are still active, while JFreeChart is relatively stable and does not change very often. We checked out major versions of the subject systems from SourceForge.net Subversion (SVN) repositories. There are 124 versions in total for three subject systems. Table I summarizes basic statistics about the three subject systems used in our study.

B. Deviation trends shown by subject systems (Q1)

To answer the first research question, we calculated *SiMo* metrics for each pair of consecutive versions of a subject system and render the change of *SiMo* metrics in a line chart. As we have three types of *SiMo* metrics (S_{Pkg_St}, S_{Pkg_Se}, S_{St_Se}), we obtained three curves for each subject system. Figure 2, Figure 3 and Figure 4 present our analysis results for JFreeChart, JHotDraw and JEdit, respectively. To facilitate the understanding of the deviation trends, we also overlaid lines of code (LOC) and lines of comment (LOCom) metrics in the figures. In each of these figures, the X axis represents software versions, the left Y axis represents the *SiMo* metrics between different modularity views, and the right Y axis represents the LOC and LOCom metrics of the system.

The first interesting observation is that different subject systems present different characteristics in their deviation trends of *SiMo* metrics. For example, the *SiMo* metrics of JFreeChart vary greatly in the beginning (about one fifth of its lifespan, from v0.5.6 to v0.9.0), and then become smooth during the rest of its lifespan, while JHotDraw shows quite smooth curves over time. JEdit presents yet another different characteristic; its modularity view similarities keep changing over its lifespan, especially the similarities between Package View and Structural Cluster View (S_{Pkg_St}) and between Structural Cluster View and Semantic Cluster View (S_{St_Se}).

The different characteristics of the deviation trends of *SiMo* metrics show different evolution stages. In JFreeChart (Figure 2), for example, three main stages are identified. We checked our speculation by a closer analysis of the release notes, change logs and UMLDiff results.

At the very beginning (versions 0.5.6 till 0.9.0), the *SiMo* metrics between different modularity views changed dramatically. This indicates a *chaotic stage*: the software was built from scratch and actively under development, and developers focused on adding new features without much consideration on maintaining a "good" modularity. Consequently, the consecutive versions show quite different *SiMo* metrics between different modularity views. Furthermore, most likely because the system was built from scratch, developers have more freedom to change the desired modularity as they wish. Therefore, the modularity views may undergo dramatic changes between versions.

Due to the changes introduced in the chaotic phase, JFreeChart might get less and less extensible and maintainable and then entered an *adjustment phase* (version 0.9.0 till 0.9.21) in which JFreeChart has been actively restructured at the same time of feature addition and extension. During this *adjustment phase*, the *SiMo* metrics between different modularity views changed dramatically from one version to another. After the version 0.9.21, the changes of the three *SiMo* metrics become similar. This indicates that a reasonable balance was achieved between the structural perspective and semantic perspective of the package organization of the system.

After the adjustment stage, JFreeChart entered a relatively stable stage – the *fine-tuning stage*. This stage may indicate that the system was mature and functional. As a result, the focus shifted from adding new features and restructuring existing ones to mainly small enhancements and bug fixes. As shown in Figure 2, from version 1.0.0.pre1 on, JFreeChart became stable. In other words, it indicates that the package structure of the system is now considered well organized and new features are rarely introduced to the system. This is evident in JFreeChart release notes: JFreeChart underwent no major changes since version 1.0.13.

TABLE I. SUBJECT SYSTEMS

Projects	Time	# Versions	# Packages	# Source Files	KLOC	KLOCom.	Source file restrictions
JFreeChart	2000/11/27-2009/4/20	50	5-63	86-805	8.0-107.4	5.9-107.6	Restricted to the source folder
JHotDraw	2000/10/13-2011/1/6	14*	7-41	128-429	9.3-55.9	4.6-29.2	Restricted to the source folder; excluding test cases and sample files
JEdit	2001/11/05-2011/2/3	60	11-29	138-503	29.2-105.8	10.4-48.4	Restricted to JEdit core

* We ignored JHotDraw versions 7.0.1 to 7.0.6 since these versions are not available

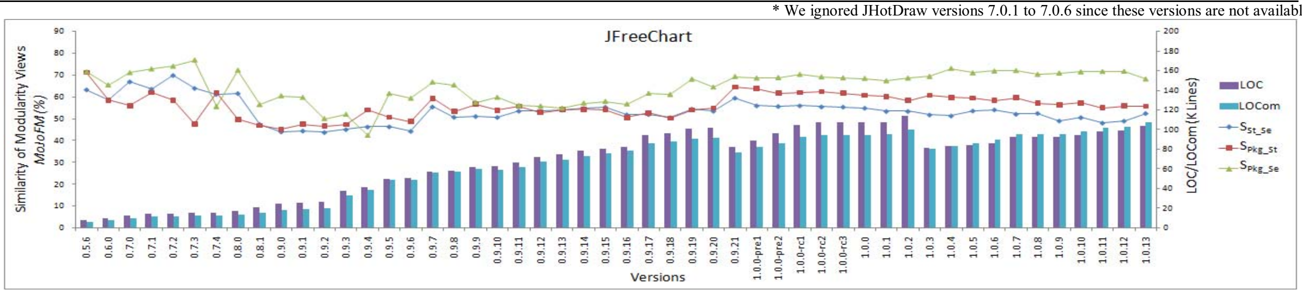

Figure 2. Similarity of modularity views and system size of JFreeChart

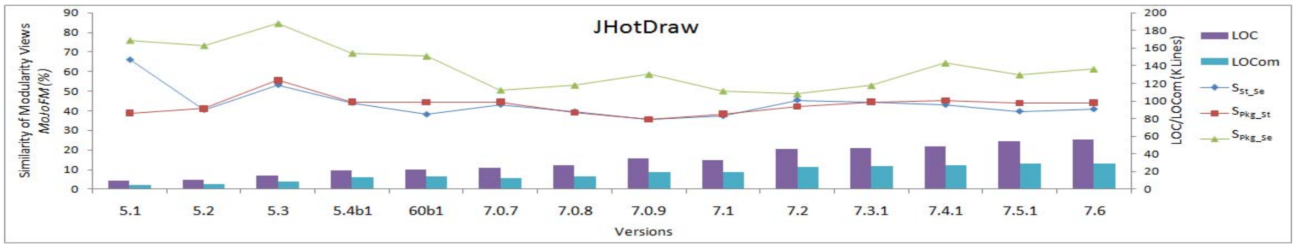

Figure 3. Similarity of modularity views and system size changing of JHotDraw

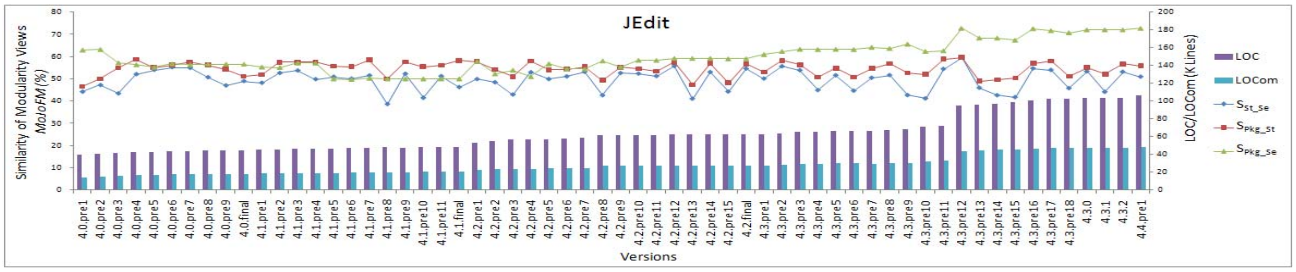

Figure 4. Similarity of modularity views and system size changing of JEdit

The second interesting observation is that the *SiMo* metrics are stable in general, with some periodical changes. This indicates that, although the high similarity between modularity views represents better design, the subtle differences between the modularity views cannot be overlooked. The similarity between modularity views cannot be monotonously rising, simply because these modularity views are from different perspectives and the designers have to continuously balance these aspects to obtain good quality of the system.

C. Understanding deviation trends (Q2)

Among the 27 patterns of deviation trends mentioned in Section IV, we find that some are closely related to maintenance actions and frequently found in the evolution history. Table II shows some deviation trend patterns found in the evolution history, together with our intuitive understandings in terms of maintenance actions.

The deviation trend patterns with our intuitive understandings have covered 65% evolutions (79 out of total 121), which is somehow representative in the whole evolution history. In order to evaluate our intuitive understandings with deviation trend patterns, we look for maintenance evidences in the *release notes* provided by software developers and the *source code changes* obtained in two versions. These two types of evidences compensate each other. Release notes express explanations of software maintenance from developers' perspective at a relatively higher abstraction, while source code differences between versions show detailed factual changes that developers have made to the software.

TABLE II. TYPICAL DEVIATION TREND PATTERNS IN OUR SUBJECT SYSTEMS

Typical deviation trend pattern			JFC	JHD	JE	Total	Intuitive understanding	Explanation of the intuitive understanding
S_{Pkg_St}	S_{Pkg_Se}	S_{St_Se}						
↘	↘	↘	5	2	2	9	Low quality evolution	All SiMo metrics drop. Maintainers have made casual or irresponsible changes that degrade the modularity of the project. This is probably due to lack of a thorough consideration of the structure of the project.
↗	↘	DNC	4	3	3	10	Structural biased evolution	SPkg_St rises but SPkg_Se drops. Maintainers have made dramatic structural changes, ignoring some semantic information when restructuring the software.
↘	↗	DNC	9	2	5	16	Semantic biased evolution	SPkg_Se rise but SPkg_St drop. Maintainers may have been focused on clarifying semantics in packages or improving semantic consistency within packages.
→|↗	→|↗	→|↗	14	3	27	44	Steady-going evolution	The software system has been well maintained. Modularity remain unchanged or improved. This is a good signal for high quality evolution.
Others			17	3	22	42		
Total			***49***	***13***	***59***	***121***		

To gather source code changes, a software difference tool, UMLDiff [36], is used for source code analysis. We run UMLDiff to find which kinds of changes occur in neighbor versions and how many of them have taken place. There are 36 types of source code differences that can be identified by UMLDiff (e.g. "add call in/out", "remove call in/out", "add data type", "remove data type", "renaming identifiers", etc. [37]). Among them only 33 are considered relative to affecting software structure or semantics ("visibility up", "visibility down", and "parameter list order change" are not considered). We categorize these types into three higher-level maintenance actions: 1) a few changes to the system; 2) dramatic changes to the system; and 3) renaming identifiers. The first two actions may contribute to both structure and semantics, while the third is a typical semantic maintenance action.

Another information source is the release notes and change logs. We specifically chose Java projects for which these materials are available. Eleven typical higher-level maintenance actions in the release notes/change logs are considered relative to our evolution patterns. Such information is usually not observable by direct source code analysis.

A guideline for relating the collected evidences to evolution patterns is listed in Table III. The evolution patterns are the inferred conclusions with our analysis to the deviation trends of the different modularity views. These patterns should be supported by one or more higher-level maintenance actions. In Table III, the meanings of symbols are as the following: a '++' means that the evidence (row) *strongly* supports the corresponding evolution pattern (column), a '+' means *some* support; a '−−' means that the evidence (row) has a *strongly* negative influence on the corresponding evolution pattern (column), a '−' means *some* negative impact, and a '○' means neutral.

We invited five software professionals to be our valued evaluators. All of the evaluators are students from the School of Computer Science of Fudan University. Two of them are PhD students and the others are master-level students. The programming experience of this group ranges from 4 to 8 years, with a median of 6 years. And two evaluators have software industry experience above 2 years.

Table III is used for our guest evaluators to evaluate how well the evidences support the intuitive understandings. In our experiment, the professionals were given the projects, the curves of SiMo indicators and Table III, and asked to look into the release notes, change logs and UMLDiff results to collect evidences for each evolution individually.

TABLE III. MAPPING TABLE TO RELATE EVIDENCES AND EVOLUTION PATTERNS

Evid. Src.	Higher-level maintenance actions	Intuitive understanding			
		Low quality	***Structural Biased***	***Semantic Biased***	***Steady-going***
Release Notes & Change Logs	Add numerous new features at one time	++	○	○	−
	SVN Commit without comments	++	○	○	−
	Relevant change logs are not found	++	○	○	−−
	Relevant Release notes are not found	++	○	○	−−
	Package-level restructuring (e.g. adding/deleting/merging /splitting packages)	○	++	+	○
	Program refactoring (e.g. adopting design patterns, extracting methods, etc.)	○	++	+	○
	Add or update comments in source files	○	−	++	+
	API additions, removes, or changes	○	++	+	○
	Update Javadoc	○	−	++	++
	Update release notes, change logs	−	○	○	++
	Only bug fixes and enhancements	−	○	○	++
	Main release number updated, or final release number is created	−	○	○	++
UMLDiff	Few changes to the system	−	+	+	++
	Dramatic changes to the system	++	+	+	−
	Renaming identifiers	○	−	++	○

First, the deviation trend of an evolution is identified to decide the intuitive understanding (column in Table III). Then, the software professionals search for evidences listed in rows in Table III. If evidences are found, a score is added according to the symbol in the corresponding cell. A

Weighted Score Method (WSM, also used in Galster's work[38]) is adopted, where '−−', '−', '○', '+', '++' stands for -2, -1, 0, 1, 2, correspondingly. After that, the scores of all evidences for the intuitive understanding are summed up for final evaluation where a Likert scale is used. The conclusions are categorized in five levels: (1) Strongly Disagree, if *summed score<-2*, (2) Disagree, if *-2 ≤summed score <0*, (3) Unsure, if *summed score=0*, (4) Agree, if *0< summed score ≤2*, and (5) Strongly Agree, if *summed score>2*. The final summed score shows the trueness of our modularity-view-based understanding of software evolution. Note that the evaluators are not required to review the source code files manually, as this work is tedious and time-consuming, and very likely to affect the evaluators' moods and reduce the quality of their evaluations.

Table IV shows the number of the evaluators' responses along the Likert scale. The evaluators *agree*d or *strongly agree*d with our hypotheses on 70 evolutions (out of a total 79, >88%), which confirmed partly the reasonability of our hypotheses. All *disagree* cases are those versions with few change logs or release notes but UMLDiff shows dramatic changes. Also, three near-by evolutions (JFreeChart v0.9.16 to v0.9.17, v0.9.18 to v0.9.19, v0.9.20 to 0.9.21) involving six neighbor versions show that, during a certain period of time, developers may "forget" to write release notes. We believe that these cases are allowable in real development. Further discussions on periodical analysis on the deviation trends are in the next subsection.

TABLE IV. EVALUATION RESULT OF THE INTUITIVE UNDERSTANDINGS

Intuitive understanding	S. Disagree	Disagree	Unsure	Agree	S. Agree	Total
Low quality	0	0	0	6	3	9
Structural Biased	0	0	1	6	3	10
Semantic Biased	0	0	0	5	11	16
Steady-going	0	8	0	18	18	44
Total	***0***	***8***	***1***	***35***	***35***	***79***

D. Feedbacks for evolution decisions (Q3)

1) Feedbacks on low quality evolutions

As the target systems in our experiments are publicly accepted as well maintained systems, there should be few casual maintenance actions. Our experiments confirmed this conjecture as there are only 9 casual evolutions in 121 evolutions.

All of these casual actions were confirmed by the evaluators. In JFreeChart (Figure 2), we find all casual maintenances occurred between earlier versions, when the project was not stable yet. In JHotDraw and JEdit (Figure 3 and 4), this situation spans all the history. We believe this is reasonable, because JFreeChart is an inactive project that is quite stable in recent years, while JHotDraw and JEdit are still quite active and not stabilized yet. The recent existence of "casual maintenance" in JHotDraw and JEdit reflects this fact.

A typical example of casual maintenance is the evolution from version 5.3 to version 5.4b1 in project JHotDraw, as can be seen in Figure 3. The release note for version 5.4b1 says:

"...It has not been extensively tested but is aimed to give developers access to a more recent version of JHotDraw than 5.3....
...The release includes numerous new features...
...Finally, developers who require a more stable release or are not willing to bear with some bugs and exceptions should not use this beta release..."

As we can see in these notes, the developers faithfully recorded that the beta version 5.4b1 was not stable and not thoroughly tested. In the next version (v6.0b1), the situation did not take a favorable turn. We believe that such tentative, unorganized modifications should not be encouraged.

Low quality evolution is a hint for quality declining. If low quality evolution occurs, it is probably not suitable to release a new version and developers are suggested to double-check modified modules for any undesirable purposes.

2) Feedbacks on structure biased evolutions

Structure biased evolution features a significant rising of S_{Pkg_St} and a declining of S_{Pkg_Se}. A typical example is between versions 4.0.pre2 and 4.0.pre3 of JEdit. In the release note of 4.0.pre3, we found 9 primary new features and a long list of enhancements and bug fixes. In the results provided by UMLDiff, we also found that 4.0.pre3 added numerous data types (389, median 75), objects (492, median 97.5) and method parameters (194, median 41.5) but removed much less ones (94 data types, 110 objects, and 91 method parameters). This is a typical function-expanding evolution. A similar situation can be found in the next evolution, from 4.0.pre3 to 4.0.pre4, and several other evolutions with the same deviation trend pattern.

Such a pattern usually infers dramatic structural changes to the software, but we find it *not* necessarily true that dramatic changes of structure will significantly influence the similarity between *Pkg* and *St* (S_{Pkg_St}). A representative example is the evolution from JHotDraw6 (v6.0b1) to JHotDraw7 (v7.0.7). Getting insight into the logs, we found most of the changes were mostly structural:

"...JHotDraw7 is a major departure of JHotDraw – Only the cornerstones of the original architecture remain. The API and almost every part of the implementation have been reworked to take advantage of the Java 2 SE 5.0 platform...
...Added new package org.jhotdraw.application...
...Moved all packages from ch.randelshofer to org.jhotdraw...
...Reorganized package structure..."

However, the changes are not reflected in Figure 3. The S_{Pkg_St} was almost stable (with a little increasing) between v6.0b1 and 7.0.7. It is very likely the case that, when a deep restructure of a project is to happen, the new version may have a completely different structure. Meanwhile, our approach does not track the structure between versions, but measures only similarity between different views instead of different versions. Therefore, even if the changes are numerous, the metrics may not be so sensitive.

3) Feedbacks on semantic biased evolutions

One typical example of semantic bias evolution type is the evolution of JFreeChart between version 0.9.4 and 0.9.5. Some evidences were found in the release notes that support our understanding, such as "*lots of Javadoc updates*". Meanwhile, there are also some logs for structural adjustment, such as "*created separate packages for the axes (com.jrefinery.chart.axis), plots (com.jrefinery.chart.plot) and renderers (com.jrefinery.chart.renderer)*". Obviously, we cannot clearly separate the semantic and structure adjustment from each other. Since these two kinds of actions are often intertwined with each other during the evolution, the boundary between them, if exist, is ambiguous. Another example is hidden in the only *unsure* response of structural biased evolution type (JFreeChart versions 0.7.3 to 0.7.4). In this evolution, although the deviation trend showed a structural biased evolution, some evidences that support semantic biased evolution was found, such as "…*Various Javadoc comment updates*…" in the v0.7.4 release note. Therefore, whether the curve will present a semantic biased evolution is largely related to both structural changes and internal doc updates, due to the strong relationship between semantic and structural adjustments.

4) *Feedbacks on steady-going evolutions*

There are two extreme cases of steady-going evolutions, according to our intuitive understandings. One is that all *SiMo* indicators rise. Typical examples of this case are the evolutions of project JFreeChart from version 0.9.20 to version 0.9.21 and the evolution of JEdit from version 4.3.pre11 to version 4.3.pre12. In these evolutions, few changes were made to the systems, and the release notes and change logs often have detailled descriptions about the changes applied to the systems. This reflects the maintainers did these changes very carefully.

The other extreme case is that all *SiMo* indicators hold. The period between versions 0.9.21 and 1.0.1 of JFreeChart is a typical example. UMLDiff showed few changes and release notes recorded few new features but only some bug fixes or minor enhancements. This reflects a comparatively stable phase of software evolution. In our empirical study, the evaluators agreed or strongly agreed 36 evolutions (about 81%) out of total 44 indicated steady-going evolutions. The other 8 evolutions get *disagree* responses. The common characteristic of these evolutions is that there are dramatic changes to the system without accompanying carefully maintained documentation. To check the reliability of these disagree responses, we further manually checked the source codes of the relevant versions of these evolutions, and finally found that, as the source codes were changed, a lot of maintenance changes were applied to the embedded comments. But unfortunately, this information was neither captured by UMLDiff nor recorded in the release notes or change logs. Embedded comments in source code should be important evidence but it is not feasible to find this evidence manually in the source. How to gather required evidence remains an open question for the evaluation.

VI. THREATS TO VALIDITY

A. *Internal validity*

In our empirical study, we largely depend on evaluators' software development experiences to get the evaluation results. Also, the scoring criterion (the -2 to +2 scale) is quite simplified. To minimize the threats, we introduced concrete and objective guidelines for the evaluators. Some evaluators worked collaboratively on the same project to eliminate random errors. After all, it would be definitely a good idea to find someone familiar with the project's evolution history and intensions to be our evaluator. As to the scoring criterion, although more complicated scoring rules can be invented, the five-level Likert approach has been proved to be effective. Inventing a complex scoring rule might reflect some truth of evolution intensions, but may also confuse the evaluators and bring more arbitrary issues.

B. *External validity*

Our study is based on only three open source Java projects. Although the projects are representative as they are typical in recent software development community, the particular evolution history of the projects may not be applicable to other projects. To minimize the threats, we carefully chose the projects that have a comparatively long evolution history with almost complete maintenance record. We do not expect to fully conclude software quality by observing the deviation trends but hope to provide a different way to evaluate possible evolution trends and quality during software maintenance. Also, commercial software systems would be a future work for us to evaluate and generalize our approach and experiences.

C. *Construct validity*

In our approach, the modularity views are extracted based on existing techniques. The reliability of program analysis tools and software clustering algorithms affects the result of the extraction process. To minimize the threats, we tried several approaches for software clustering before conducting a systematic empirical study. We finally adopted a widely used algorithm to reflect the state-of-art of clustering technique. Typically, there are two criteria for evaluating the quality of clustering results, namely *Authoritativeness* (the produced partition should be very similar to authoritative partition) and *Non-extremity distribution* (clusters in the produced partition should be neither *black holes* nor *gas cloud* [6]). However, an authoritative partition is often hard to find. Furthermore, we investigate the target systems from different modularity views; this makes the task even more complex. We think non-extremity clusters may reveal the reasonability of the partition. This criterion was assessed by NED (non-extreme distribution) measure [22]. In our presented experiments, more than three-quarters of the NED values are above 0.72 for structural clustering and 0.85 for semantic clustering, showing acceptable clustering results.

VII. CONCLUSION AND FUTURE WORK

In this paper, we propose an approach for monitoring design quality evolution of software systems in long-term evolution by analyzing the deviation trends of different modularity views. Currently, we consider three modularity views, namely the desired modularity view reflected by the package structure, and two implied modularity views extracted by structural and semantic clustering. Based on the approach, we conduct an experience study to evaluate our approach on the one hand, and identify typical deviation trend patterns on the other hand. We find that the deviation trends of different modularity views largely indicate the status of design quality evolution, especially the trends of quality degradation. And the continuous monitoring of deviation trends provides useful feedback for the future evolution decisions.

We have noticed that our hypothesis and approach are still to be further confirmed with wider range of software projects. In order to overcome the difficulty of the lack of evolution intensions and comprehensive maintenance logs, we are trying to extend our experience study to internal and external projects to increase the generality of our conclusion. In the future work, we will also involve more modularity views and provide more comprehensive deviation trend monitoring for evolution decisions.

ACKNOWLEDGMENT

The work presented in this paper is supported by Natural Science Foundation of China under grants 90818009, 60903013.

REFERENCES

[1] F. Van Rysselberghe, S. Demeyer, "Studying software evolution information by visualizing the change history," in ICSM, 2004, 328-337.

[2] Z. Xing, E. Stroulia, "Analyzing the evolutionary history of the logical design of object-oriented software," in TSE, 2005. 31(10): 850-868.

[3] S. Huynh, Y. Cai, Y. Song, K. Sullivan, "Automatic modularity conformance checking," in ICSE, 2008, 411-420.

[4] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," In Classics in Software Engineering, Edward Nash Yourdon (Ed.). Yourdon Press, Upper Saddle River, NJ, USA, 1979, 139-150.

[5] S. Wong, Y. Cai, M. Kim, M. Dalton, "Detecting software modularity violations," in ICSE. 2011, 411-420.

[6] N. Anquetil, T. C. Lethbridge, "Experiments with clustering as a software remodularization method," in WCRE, 1999, 235-255.

[7] A. Kuhn, S. Ducasse, T. Girba, "Enriching reverse engineering with semantic clustering," in WCRE, pp. 10 p, 7-11 Nov. 2005.

[8] T. Mens, S. Demeyer, "Future trends in software evolution metrics," in IWPSE '01, 2001, 83-86.

[9] M. M. Lehman, L. A. Belady, "Program Evolution Process of Software Change," Academic Press London and New York, 12-26, 1985.

[10] M. Godfrey, Q. Tu. Qiang, "Growth, Evolution, and Structural Change in Open Source Software," in IWPSE '01, 2001, 103-106.

[11] J. F. Ramil, M.M. Lehman, "Defining and applying metrics in the context of continuing software evolution," in METRICS, 2001, 199-209.

[12] Z. Xing, E. Stroulia, "Understanding phases, styles of object-oriented systems' evolution," in ICSE, 2004, 242-251.

[13] A. Mockus, L. G. Votta, "Identifying reasons for software changes using historic databases," in ICSM, 2000, 120-130.

[14] W. P. Stevens, G. J. Myers, L. L. Constantine, "Structured design," IBM Systems Journal, vol. 13, 1974, 115–139.

[15] F. Brito E Abreu, M. Goulao, "Coupling and cohesion as modularization drivers: are we being over-persuaded?," in CSMR, 2001, 47-57.

[16] H. Abdeen, S. Ducasse, H. Sahraouiy, I. Alloui, "Automatic Package Coupling and Cycle Minimization," in WCRE, 2009, 103-122.

[17] A. Marcus, D. Poshyvanyk, "The conceptual cohesion of classes," in ICSM, 2005, 133-142.

[18] D. Poshyvanyk, A. Marcus, "The Conceptual Coupling Metrics for Object-Oriented Systems," in ICSM, 2006, 469-478.

[19] H. Kagdi, M. Gethers, D. Poshyvanyk, M. L. Collard, "Blending Conceptual and Evolutionary Couplings to Support Change Impact Analysis in Source Code," in WCRE, 2010, 119-128.

[20] J. I. Maletic, N. Valluri, "Automatic software clustering via Latent Semantic Analysis," in ASE, 1999, 251-254.

[21] A. Kuhn, S. Ducasse, T. Girba, "Semantic clustering: Identifying topics in source code," in IST Journal, 2007. 49(3): 230-243.

[22] G. Scanniello, M. Risi, G. Tortora, "Architecture Recovery Using Latent Semantic Indexing, K-Means: An Empirical Evaluation," in SEFM '10, 2010, 103-112.

[23] G. Bavota, A. D. Lucia, A. Marcusy, R. Oliveto, "Software Re-Modularization Based on Structural and Semantic Metrics," in WCRE, 2010, 195-204.

[24] T. A. Wiggerts, "Using clustering algorithms in legacy systems remodularization," in WCRE, 1997, 33-43.

[25] J. Wu, A. E. Hassan, R. C. Holt, "Comparison of clustering algorithms in the context of software evolution," in ICSM, 2005, 525-535.

[26] J. Kothari, T. Denton, A. Shokoufandeh, S. Mancoridis, A. E. Hassan, "Studying the Evolution of Software Systems Using Change Clusters," in ICPC, 2006, 46-55.

[27] W. Zhihua, V. Tzerpos, "An effectiveness measure for software clustering algorithms," in IWPC, 2004, 194-203.

[28] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, R. Harshman, "Indexing by Latent Semantic Analysis", Journal of the American Society for Information Science, vol. 41, 1990, 391-407.

[29] Weka, http://www.cs.waikato.ac.nz/~ml/weka/index.html, 2011

[30] V. Tzerpos, R. C. Holt, "MoJo: a distance metric for software clusterings," in WCRE, 1999, 187-193.

[31] W. Zhihua, V. Tzerpos, "An optimal algorithm for MoJo distance," in IWPC, 2003, 227-235.

[32] JFreeChart, http://www.jfree.org/jfreechart/, 2011

[33] JHotDraw, http://www.jhotdraw.org/, 2011

[34] JHotDraw, http://www.randelshofer.ch/oop/jhotdraw/, 2011

[35] JEdit, http://www.jedit.org/, 2011

[36] Z. Xing, E. Stroulia, "UMLDiff: An Algorithm for Objectoriented Design Differencing," in ASE, 2005, 54-65.

[37] Z. Xing, E. Stroulia, "API-Evolution Support with Diff-CatchUp," in ICSE, 2007, 818-836.

[38] M. Galster, A. Eberlein, M. Moussavi, "Systematic selection of software architecture styles," IET Software, 2010, 4(5): 349-360.

[39] ISO9126, "Information Technology - Software Product Evaluation - Software Quality Characteristics and Metrics," Geneva, Switzerland: International Organization for Standardization.

[40] D. V. Steward, "The Design Structure System: A Method for Managing the Design of Complex Systems," IEEE Transactions on Engineering Management, 1981, 28: 71–74.

[41] M. W. Godfrey, T. Qiang, "Evolution in open source software: a case study," in ICSM, 2000, 131-142.

[42] S. Ali, O. Maqbool "Monitoring software evolution using multiple types of changes," in ICET, 2009, 410-415.

[43] S. Demeyer, S. Ducasse, and M. Lanza, "A hybrid reverse engineering approach combining metrics and program visualization," in WCRE '99, 1999, 175-186.

[44] S. Husein, A. Oxley, "A Coupling and Cohesion Metrics Suite for Object-Oriented Software," in ICCTD '09, 2009, 421-425.

[45] H. P. Breivold, M. A. Chauhan, M. A. Babar, "A Systematic Review of Studies of Open Source Software Evolution," in APSEC, 2010, 356-365.