JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 36, 53-73 (2020)
DOI: 10.6688/JISE.202001_36(1).0004

Design Pattern Analysis with Software Evolution Data

NIEN-LIN HSUEH
Department of Information Engineering and Computer Science
Feng Chia University
Taichung City, 407 Taiwan
E-mail: {nlhsueh}@fcu.edu.tw

The influences of design patterns on software quality have attracted increasing attention in the area of software engineering, as design patterns encapsulate valuable knowledge to resolve design problems, and more importantly to improve the design quality. Since most design patterns are designed to enhance the maintainability, a system with such design patterns is expected to have lower maintenance load in its further evolution. However, sometimes design patterns are over applied or mis-used in many systems, which will cause another maintenance problem or impair the system performance. Therefore, lots of researchers proposed their approaches to evaluate the quality of design patterns or their deployment. However, there is no approach taking the software evolution into concern, even it is the major issue a design pattern addresses. In this paper, we propose a new approach to formulate the evaluation of a design pattern's utilization using the evolution data. We also conduct our approach to 11 design patterns over 15 projects which provide *software evolution data*. The analysis results show that the utilization of deployed design patterns does not have significant difference among the design patterns in the evolution of software design.

Keywords: design pattern, software evolution data, software quality, software design, open source software

1. INTRODUCTION

Software change is inevitable. Successful software systems must evolve to satisfy changing requirements. How to design a flexible architecture to accommodate changes becomes an important design problem in the area of software engineering. Recently, many design patterns have been proposed to solve such problems and improve software quality [8, 9, 26]. A growing number of practitioners have shown great interests in using design patterns towards high-quality software [4, 10, 11, 27, 34].

Although design patterns have been widely applied, its contribution is questioned and various approaches are proposed to validate the correctness of design pattern application [21, 23, 24, 32] and its value in quality improvement [1, 13, 33]. To achieve that, most of these researches analyze the programs with a given quality model [2, 13, 15, 30]. However, the analysis is conducted based on a static structural model, *i.e.*, a snapshot of the software product in its development life cycle. As we know, most patterns address

Received July 19, 2018; accepted September 17, 2018.
Communicated by Jung-Hsien Chiang.
* This research was supported by the Ministry of Science and Technolody, Taiwan, under grants MOST 106-2221-E-035-MY2.

the maintainability issues, therefore the evaluation should be based on all versions in its software evolution, not only a specific version.

In this paper, we propose an analysis method for evaluating the deployed design patterns *in software evolution*. Since most design patterns provide more flexible architecture to enhance maintainability. Such patterns should be inspected to see if they can be utilized to meet the original design purpose. As shown in Fig. 1, if a developer deploys a *Strategy* design pattern dp_1 in version v_1 and it intends to extend the *Strategy* in the future, but no *Strategy* objects are added or removed to the original design for the extension from v_1 to v_{10} in the evolution. We may consider the application of the pattern is less utilized in the evolution. On the other hand, Fig. 2 illustrates a case that the design patterns are more utilized. From v_1 to v_2, a *Strategy* class is removed; from v_2 to v_n, another *Strategy* class is added. The *Strategy* design pattern contributes to the design evolution since it allows us to embrace a lot of design changes by extending rather than modification. The expected utilization represents the hot spot for change in the evolution.

We perform experiments on 11 design patterns over 15 open source software (OSS) projects which have totally 296 software evolutions. The open source software source code and the associated data can be archived in a public version control system which provides a transparent way for researchers to evaluate software quality. SourceForge has provided 3.7 million registered users to create powerful software in over 430,000 projects until the first quarter of 2014. The huge amount of open source software can provide world-wide software developers valuable design experiences. We conduct the experiments on OSS systems to validate our approach and our research's objectives include:

- Proposing an approach to evaluate the utilization of design patterns using the software evolution data;
- Developing a tool to support the proposed approach;
- Applying our approach to some well-known open source systems which provide software evolution data; and
- Investigating whether the utilization of deployed design patterns are correlated with the software projects or with the types of design patterns.

The remainder of this paper is structured as follows: Section 2 discusses background work. Section 3 introduces our approach to measure the utilization of the design patterns in software evolution. In Section 4, we demonstrate the system design approach and report our experimental results. Our analysis and discussion are presented in Section 5. Finally, conclusion remarks are given in Section 6.

2. BACKGROUND WORK

A number of researchers have applied different methods to evaluate the design patterns from different motivation. In section 2.1, we introduce and compare these re-

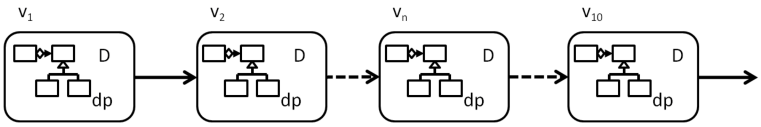

Fig. 1. Less utilized design pattern application in software evolution.

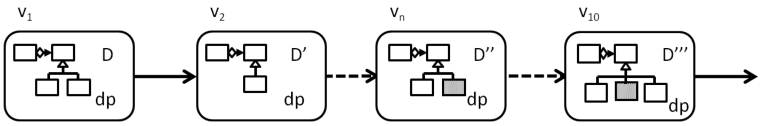

Fig. 2. Utilized design pattern application in software evolution.

searches. In section 2.2, we introduce the two pattern detection work since they are important background techniques for the pattern evaluation researches.

2.1 Evaluation of Design Pattern Application

Gamma *et al*. [9] propose twenty-three design patterns which are aimed for solving software design problems. They advocate that design patterns can reduce system complexity by naming and defining abstractions. Diverse techniques are applied to evaluate the quality of deployed design patterns in related studies.

Software metric is a measure of defined property of a piece of software and a common way to evaluate software quality. Huston [15] describes the effects of some patterns on object-oriented software metrics. His study concludes that different metrics may produce diverse and sometimes conflicting results on programs deployed with design patterns. Hsueh *etal*. [13] propose a validation approach to help developers check if a design pattern is well-defined. A quantitative method is proposed to measure the effectiveness of the quality improvement to meet their functional and quality requirements.

Controlled experiment is also a way to evaluate the role of design patterns in software design. Prechelt *et al*. [23] test whether the use of some specific design patterns is helpful for participants with different backgrounds. Prechelt *et al*. [24] study the problem of quality of design patterns from another perspective. They examine whether documentation of deployed design patterns improve the functional quality in maintenance actions to perform software changes. Similar related study by Ng *et al*. [21] discusses the usage of design patterns and its importance for software maintenance.

Some researchers apply tools to examine the costs of design pattern applications. Aversano *et al*. [2] analyze how frequently the deployed design patterns are modified. Vokáč [30] tries to find a relation between the presence of specific design patterns in software and the number of defects. Izurieta *et al*. [16] introduce the notion of design pattern grime and perform a study of the effects of decay on three open source software.

Recently, there are papers which survey the effective usage of deployed design patterns. Ng *et al*. [20] study whether maintainers utilize deployed design patterns and the commonly performed tasks when they do the maintenance. Their study conclude that the delivered code is significantly less faulty than the code deployed without utilizing design patterns regardless of the type of tasks performed by maintainers. Cheng *et al*. [32] investigate how extensive usage of design patterns has been subjected to empirical study by conducting a systematic literature review in the form of mapping study. Cheng *et al*. [33] investigate the usage of design patterns which expert users consider as useful for software development and maintenance. They conclude that three patterns are regarded as useful and one quarter of patterns gain lower approval.

Hsueh *et al*. [12] propose an analysis method for the effectiveness of deployed design patterns in software evolution. Their study propose two different measurement ways for the application of design patterns including occasion and effectiveness analysis. A web-based pattern effectiveness analyzer is developed and an open source project JAXE is

Table 1. Summary of researches of evaluation of design patterns.

Research	Motivation	Method	Code Analysis	Subjects
Huston [15]	Analyze the compatibility between design patterns and design metrics	Analysis	Yes	Design Patterns
Hsueh [13]	Validate if a design pattern well-designed	Quantitative Method	No	Design Patterns
Izurieta and Bieman [16]	Examine the extent to which software designs decay, rot, and accumulate grime	Tools	Yes	Software
Zhang and Budgen [32]	Investigate how extensively the use of design patterns and how and when they can provide effective mechanism for knowledge transfer	Survey	No	Research Papers
Zhang and Budgen [33]	Identify which design patterns are considered to be useful by experienced users	Survey	No	Human
Prechelt *et al*. [23]	Investigate whether using design patterns is beneficial even the actual design is simpler	Controlled Experiments	No	Human
Prechelt *et al*. [24]	Test if the design patterns in the program code are documented explicitly can help the maintainer	Controlled Experiments	No	Human
Aversano *et al*. [2]	Analyze how frequently patterns are modified, to what changes undergo and what classes co-change with the patterns	Tools	Yes	Software
Vokáč [30]	Investigate if design with properly applied patterns reduces defect frequency	Self-developed Tool	Yes	Software
Ng *et al*. [20]	Analyze how maintainers utilize the design patterns to complete anticipated changes in software	Experiments	Yes	Software
Ng *et al*. [21]	Identify factors with contributions to the maintenance of programs with deployed design patterns	Experiments	No	Human
Hsueh [12]	Evaluate the effectiveness of deployed design pattern in software evolution	Self-developed Tool	Yes	Software

analyzed. From the experimental results, they find that most of the patterns are effective in the late stage of their software evolution.

We summarize the researches and classify them by four criteria in Table 1. The first criterion, called **motivation**, describes briefly the research topic of each paper. Then, we can observe the studies of design pattern usage from different perspectives. The second criterion, called **method**, lists the technique applied in each paper research. The method reflects the way that researchers used for identifying the motivation of their studies. The third criterion, called **code analysis**, distinguishes the method applied by each study if it analyzes the program source code or not. The fourth criterion, called **subjects**, specifies the research targets of each study. The subjects among the listed researches are design patterns, software, research papers and human being.

From the summary, we observe that there is still a dearth of research which evaluates the deployed design patterns over software evolution by an automatic method. In this paper, we develop an automatic evaluation tool to assess the utilized usage of deployed patterns when software evolves. This is a pioneer study of quality evaluation of deployed design patterns in software evolution.

2.2 Design Pattern Detection

In our work, we apply the pattern detection technique developed by Tsantalis *etal*. [28] to identify the deployed design patterns for evaluation. Tsantalis *etal*. [28] use similarity scoring algorithm between graph representations to automatically detect modified

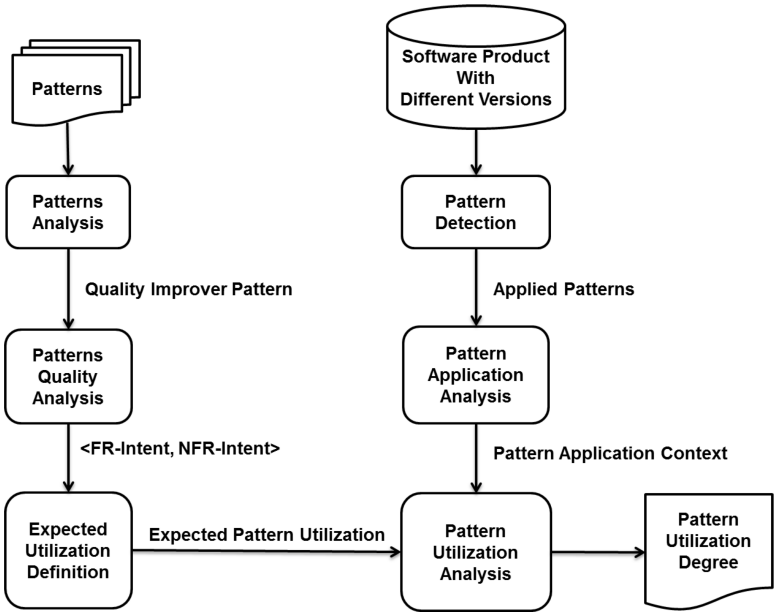

Fig. 3. Utilization evaluation of design patterns.

design patterns. Initially, they represent important aspects of the software and the design patterns static structure as graphs and matrices. Next, a graph similarity algorithm is applied to identify the instances of candidate design patterns. Dong *etal*. [6] propose a *DP-Miner* toolkit to recover instances of design pattern based on the use of matrix and weight. *DP-Miner* builds a matrix from source code instead of the graph representation to improve the accuracy of pattern detection. It can perform static behavioral analysis, but does not support pattern deviations and dynamic analysis.

Experienced developers apply design patterns in software development to solve design problems and reduce software maintenance costs. However, software systems evolve over time, increasing the chance that the design patterns in its original form will be broken. Pattern detection is a kind of reverse engineering technique for recognizing patterns from source code [19] or design [5]. Some works [6, 28] recognize patterns in the legacy system by matrix-based approach. The matrix-based approach maps all classes in the system to the rows and columns of a matrix, and the value of the corresponding cell is the relationship between each couple of classes.

3. UTILIZATION EVALUATION OF DESIGN PATTERNS

This section presents the methodology to evaluate the utilization of deployed design patterns which is shown in Fig. 3. Since not all design patterns are proposed for addressing maintenance problems, our approach starts from analyzing the design patterns. The left part of the figure is to analyze the design patterns and define the expected utilization of the pattern deployment. The right part of the figure is to perform experiments on real software projects based on the definitions of the deployed pattern utilization from the left part of this figure.

From the left part of Fig. 3, the design patterns are analyzed from different per-

spectives at first. Since not all the design patterns are expected to be utilized during the evolution, we have to classify the patterns in section 3.1. After the first step of analyis, we have identified *quality-improver* patterns and discuss them in section 3.2. When a pattern is identified as a *quality-improver*, its intent can be separated into FR-intent (functional requirement intent) and NFR-intent (nonfunctional requirement intent). The expected utilization of the *quality improver* patterns from the FR-intent and the NFR-intent are discussed in section 3.3.

After the stage of analyais shown in the left part of Fig. 3, we can define a deployed design pattern declared to be *utilized* if the original design purpose can be met as the software evolves. To perform experiments on software projects, we have to define the design pattern application context in a design which is defined as a *Pattern Application Context (PAC)* and is described in section 3.4. In section 3.5, we introduce the expected utilized application context of a pattern in a software evolution as an *Expected Pattern Utilization (EPU)* and the utilization analysis of design pattern applications is presented.

3.1 Pattern Analysis

Patterns are designed for different purposes. In this step, a design pattern is analyzed from different perspectives to see how it can facilitate design activities, handle non-functional requirement, solve design problems and resolve design conflicts [14]. Based on the analysis, design patterns are classified as *activity-facilitator* for facilitating design activities; *quality-improver* for handling non-functional requirements and improving software quality; *problem-solver* for solving design problems; and *conflict-resolver* to resolving design conflicts.

Activity-facilitator. During system design, various design activities are performed to reduce the gap between analysis models and the final executable system. A design activity is a common and recurring task for a specific objective during system design. As designing a system is not algorithmic, each involved activity is usually facilitated with heuristic knowledge or well-known patterns. A design pattern is viewed as an *activity-facilitator* if it can facilitate a specific design activity like decomposition, object allocation, access control, control flow and component composition.

Quality-improver. Non-functional requirements are not easy to handle because they are subjective, relative and interacting [7, 17]. Design patterns provide a possible way to deal with non-functional requirements since they provide solutions to satisfy functional requirements as well as better solutions to meet non-functional requirements [31]. In other words, besides providing a basic, functional solution to a problem, a design pattern plays as a *quality-improver* to offer a qualified, non-functional improvement to that solution.

Conflict-resolver. Finding the right balance for conflicting requirements is important in achieving successful software requirements and products [3, 18]. Exploring conflicts is not easy, resolving conflicts is even harder. Conflicts resulting from competing resource and divergent expectation can be resolved by specializing resource, involving agents or their behavior, or introducing an arbitrator to dispatch the resource. Conflicts arising from side effects are more difficult to handle since the involved requirements do not have a common interest they are concerned about. It is even worse that side effect conflicts occur frequently in design phase. To resolve conflicts more efficiently, a design pattern plays as a *conflict-resolver* to resolve conflicting requirements.

Problem-solver. In addition to the fundamental activities of performance, and quality and conflicting problems to deal with during object design, we also need to examine boundary conditions or possible exceptions to system reliability. Analysing design pattern from a problem view is performed to examine *how a design pattern can solve design problems and prevent possible exceptions* such as model inconsistency, data corruption or connection failure. A design pattern plays the role of *problem-solver* to solve a specific design problem. From this perspective, *Observer* design pattern is used to resolve the inconsistency problem between a set of objects (said *observers*), which have a common interest on a *subject* object. By requiring the *observers* to register on the *subject* before operating, *observers* can be notified whenever the *subject* changes its status.

3.2 Pattern Quality Analysis

After the first step analysis, the *quality-improver* patterns are identified. The identified *quality-improver* pattern's intent is separated into FR-intent and NFR-intent so as to highlight its quality contribution. The FR-intent describes what the pattern does, and the NFR-intent concentrates on the extension to the FR-intent to describe how well this pattern can contribute to quality attributes, such as reusability, maintenance, or extensibility.

Unlike a NFR-intent, which is realized in a complex structure, FR-intent is realized in a more simple structure called FR-structure [14]. Fig. 4 left part illustrate the FR-structure and its corresponding NFR-structure of *Strategy* design pattern. In Fig. 4 right part, a new role *Strategy* is defined to declare an interface for implementing algorithms. Essentially, NFR-structure is an extension of FR-structure designed to satisfy the associated NFR-intent. The extension plays an important role in helping us transfer a basic model to an extension model that is compatible with the NFR-structure.

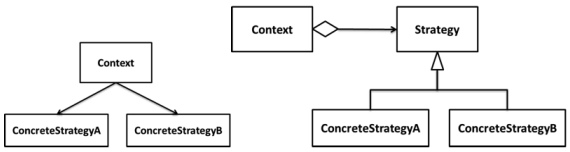

Fig. 4. FR- and NFR-structure of *Strategy* design pattern.

In this step, we verify the design patterns by examining if the structure can satisfy the NFR-intent. Since we have defined a deployed design pattern is declared to be utilized if the original design purpose can be met as the software evolves. The utilization of a design pattern is evaluated according to the degree of satisfaction of non-functional requirements. The kinds of design patterns is classified as *quality-improver*.

3.3 Expected Utilization Definition

In the previous step, the *quality-improver* patterns are examined if the structure can satisfy the NFR-intent. To verify the quality contribution of deployed design patterns in software design, the patterns should be inspected over its evolution according to their expected utilization. The expected utilization of a deployed design pattern is defined to be the improved operation over the pattern deployment evolution.

Take *Strategy* pattern as an example, the functional intent of the *Strategy* pattern is for an object class to use an algorithm to resolve a specific problem. The non-functional intent of the *Strategy* pattern is to easily replace a new one algorithm for enhancing extensibility. By observing the non-functional intent, we can identify the utilization of *Strategy*

pattern can be satisfied by adding *ConcreteStrategy* class with a new algorithm or removing *ConcreteStrategy* class with the unnecessary algorithm.

3.4 Pattern Application Analysis

In the previous section, a deployed *quality-improver* design pattern is declared to be *utilized* if the original design purpose can be met as the software evolves. In order to inspect whether the deployed *quality-improver* design patterns can meet the definitions of expected utilization. We have to define the structure of pattern application context in software design.

Pattern Application Context is to describe an application context design pattern *dp* in a design *D*. Design *D* includes a design element *element* and a participation role *r*. We define a *PAC* as:

$$PAC(dp, D) = \{< element, r >\}. \tag{1}$$

For example, we may apply a *Strategy* pattern in a design for data compression. The *Strategy* pattern contains design elements *DataCompression*, *CompressionTool*, *RAR* and *ZIP* which play the roles of *Context*, *Strategy* and *Concrete Strategy*, respectively. The *PAC* of *Strategy* pattern in a design *D* can be defined as:

$$\begin{aligned} AC(Strategy, D) = \{&< DataCompression, R_C >, \\ &< CompressionTool, R_S >, \\ &< RAR, R_{CS} >, < ZIP, R_{CS} >\}, \end{aligned} \tag{2}$$

where R_C, R_S, R_{CS} denotes the role of *Context*, *Strategy*, *Concrete Strategy* respectively.

3.5 Utilization Analysis of Pattern Application

According to the definition of expected utilization of deployed *quality-improver* design patterns and the definition of pattern application context in software design. We can determine a deployed *quality-improver* design pattern is *utilized* in software evolution if its pattern deployment can meet the expected pattern utilization.

Expected Pattern Utilization of a *quality improver* pattern is defined to describe the expected application utilization context of a pattern in a software evolution. A *EPU* is defined as:

$$EPU(dp) = \{< role, operator >\}. \tag{3}$$

For a design pattern, *role* is the element in a design pattern *dp* and *operator* is the expected operation applied to a design when conducting the pattern. The operator may be add (+) or remove (-). Take *Strategy* pattern for example, we expect to add a new component or remove an existing component without modifying other design elements. In other words, we expect to add/remove a design element which plays the *"Concrete Strategy"* role to/from the design. Thus, we can define the *EPU* of *Strategy* pattern as:

$$EPU(Strategy) = \{< R_{CS}, + >, < R_{CS}, - >\}. \tag{4}$$

Again, the R_{CS} denotes for the role of *Concrete Strategy*.

According to the definition of *PAC* and *EPU*, a design pattern is declared to be utilized when its *PAC* evolves conform to the *EPU*. We define the predicate *utilized*(*dp*,

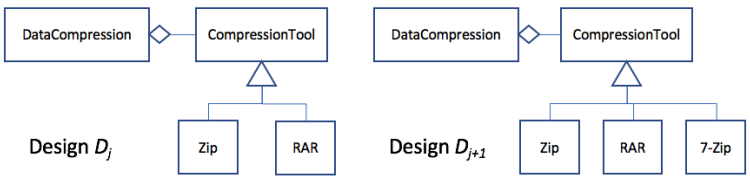

Fig. 5. Utilized pattern application.

D_{j+1}) to represent if a design pattern dp is utilized at certain version D_{j+1}. The formal definition is as below:

$$\begin{aligned} &\textit{utilized}(dp, D_{j+1}) \Leftrightarrow \\ &\exists\, D_j, \Delta(\textit{PAC}(dp, D_j), \textit{PAC}(dp, D_{j+1})) \\ &\textit{conforms to EPU}(dp), \end{aligned} \tag{5}$$

where $\Delta(\textit{PAC}(dp, D_j), \textit{PAC}(dp, D_{j+1}))$ denotes the difference between $\textit{PAC}(dp, D_j)$ and $\textit{PAC}(dp, D_{j+1})$. The design D_j and D_{j+1} deploy the same design pattern dp. As presented in Fig. 5, the design D_j deploys a *Strategy* pattern. The intent behind the application of applying a *Strategy* pattern is to provide flexible alternatives to inheritance to combine class behavior with new functionalities [9]. Adding "7-zip", a data compression tool, in a later version of D_{j+1}, is to utilize the design pattern *Strategy* to satisfy the quality requirement of the extensibility.

The utilization analysis is to calculate the percentage of utilization of a design pattern's application in a software evolution. A higher percentage means frequent pattern application and implies greater contribution to the evolution. However, it is hard to identify that the deployed application is useful or not. It relies on the developers' active intervene. The degree of utilization over a period of evolution (denoted as DoU) can be defined as:

$$DoU = \frac{\#of\,utilized\,applications}{\#of\,pattern\,evolution} \tag{6}$$

The DoU can help us understand the utilized status of design pattern application. Take Fig. 6 as an example. There are eight versions (V_1-V_8) and five design pattern instances (dp_1-dp_5) for that software. The black triangle means a design pattern is deployed at that version. "X" means that the design pattern is removed from that version. The number of utilized applications includes the counts of changes between software versions. The number of evolution is the total number of versions minus one. A design pattern instance dp_1 is deployed at V_1 and stays alive consecutively for eight versions. The number of evolution is seven and the number of utilized applications is two for (V_3-V_4) and (V_4-V_5). The DoU of dp_1 is 2/7. Pattern instance dp_2 is deployed at V_1 and stays alive consecutively for eight versions. The number of evolution is seven and the number of utilized applications is three for (V_3-V_4), (V_4-V_5) and (V_6-V_7). The DoU of dp_2 is 3/7. Pattern instance dp_3 is deployed at V_3 and stays alive consecutively for six versions. The number of evolution is five and the number of utilized applications is one for (V_4-V_5). The DoU of dp_3 is 1/5. Pattern instance dp_4 is removed after two versions. The number of evolution is two and the number of utilized application is one for (V_3-V_4). So the DoU of dp_4 is 1/2. Pattern instance dp_5 is not utilized at any version. The DoU of dp_5 is 0/5.

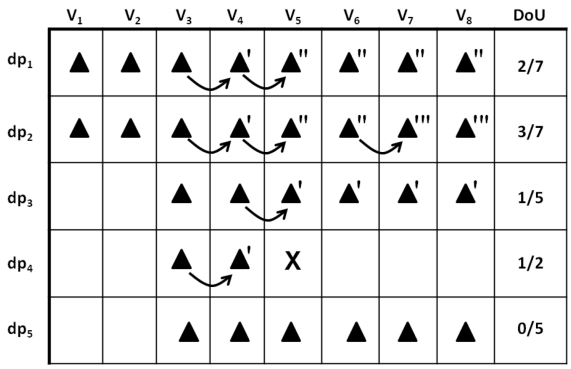

Fig. 6. Utilization analysis of five instances during eight versions.

4. SYSTEM DESIGN AND EXPERIMENTAL RESULTS

The emergence of open source software has changed the way of researches on software quality. The open source software source code and the associated data are archived in the version control system for researchers to evaluate software quality in a transparent way. We consider fifteen open source software which had developed for a long period of time and through a certain versions of release for our experiment evaluation.

4.1 Context Description

The context of this study consists of fifteen OSS projects. Table 2 lists the basic project information of these evaluated OSS projects including the project name, the number of evolution, the versions and the development time interval.

JHotDraw is a two-dimensional graphics framework for structured drawing editors that is written in Java. JHotDraw is a Java GUI framework for technical and structured Graphics. It has been developed as a "design exercise" but is already quite powerful. Its design relies heavily on some well-known design patterns. JHotDraw is a popular evaluated software in many researches [1, 21, 22, 28]. It is based on Erich Gamma's JHotDraw, which is copyright 1996, 1997 by IFA Informatik and Erich Gamma.

JUnit is also a popular evaluated project in some researches [22, 28]. JUnit is a simple framework for writing and running automated tests. As a political gesture, it celebrates programmers testing their own software. Like JHotDraw as a graphic tool, FreeMind is a premier free mind-mapping software which is written in Java language.

We randomly choose other 12 open source projects including HtmlUnit, Jaxe, JConvert, jEdit, Jena, jGnash, jMemorize, LatexDraw, PDFsam, Pixelitor, TuxGuitar and Weka. HtmlUnit is a "GUI-Less browser for Java programs", which allows high-level manipulation of web pages, such as filling forms and clicking links. Jaxe is a free Java XML editor with a configurable GUI, using XML schemas for validation and XSL for exports in HTML or XML. JConvert is a free unit conversion program that has a friendly user interface and can also be used with external applications.

JEdit is a multi-platform mature programmer's text editor which is written in Java. Jena is a Java toolkit for developing semantic web applications based on W3C recommendations for RDF and OWL. It provides an RDF API; ARP, an RDF parser; SPARQL, the W3C RDF query language; an OWL API; and rule-based inference for RDFS and OWL. JGnash is a cross platform personal finance application written in Java. JMemorize is

written in Java and uses Leitner flashcards to make memorizing facts not only more efficient but also more fun. JMemorize manages your learn progress and features categories, Unicode flashcard texts, statistics and an intuitive interface.

LatexDraw is a multi-platform graphical drawing editor for LaTeX which can be used to generate PSTricks code and directly create PDF or PS pictures. LaTeXDraw is developed in Java and runs on top of Linux, Windows, and Mac OS X. PDFsam (PDF Split and Merge) is an easy to use tool to merge and split PDF documents. The GUI is written in Java Swing and it provides functions to select files and set options.Pixelitor is a free and open source image editing software that supports layers, image effects, multiple undo. TuxGuitar is a multitrack guitar tablature editor and player written in Java-SWT. It can open GuitarPro, PowerTab and TablEdit files. Weka is a collection of machine learning algorithms for solving real-world data mining problems. It is written in Java and runs on almost any platform. The algorithms can either be applied directly to a dataset or called from your own Java code.

4.2 System Design and Application Approach

In this study, we develop a web-based tool *Pattern Utilization Analyzer* (*PUA*) to demonstrate our proposed approach. Fig. 7 shows the system architecture of our developed tool *PUA*. There are three system modules for *PUA* including version control, *PAC* and *EPU* modules. *PUA* applies a pattern detection tool, developed by Nikolaos Tsantalis [29], to detect the pattern application in software source code. The version control module is designed for managing the collected source code. The *PAC* module is designed for collecting the pattern application context for different design patterns after applying design pattern detection tool. The *EPU* module is for verifying if the *PAC* conforms to the *EPU* between consecutive versions of each OSS project.

In the following, we introduce the processes to apply our tool and our approach. The

Table 2. Evaluated OSS projects information.

Project Name	# of Evolution	Versions	Time Interval
FreeMind	16	0.0.2-0.9.0	June 2000 to February 2011
HtmlUnit	28	1.0.0-2.9.0	May 2002 to August 2011
Jaxe	38	1.0-3.5	June 2002 to June 2011
JConvert	9	1.0.0-1.1.0	August 2007 to May 2011
JEdit	27	2.3-4.5.1	January 2000 to March 2012
Jena	17	2.0-2.6.4	August 2003 to December 2010
JGnash	26	1.0.1-1.10.5	January 2003 to March 2006
JHotDraw	8	5.2-7.6	February 2001 to January 2011
JMemorize	12	0.7.0-1.3.0	October 2004 to March 2008
JUnit	19	3.4.0-4.10.0	December 2000 to September 2011
LatexDraw	24	1.0.2-2.0.8	January 2006 to March 2010
PDFsam	14	1.0.0-2.2.1	August 2007 to July 2012
Pixelitor	34	0.1.0-1.1.2	November 2009 to November 2010
TuxGuitar	11	0.2-1.2	June 2006 to November 2009
Weka	13	3.0.1-3.7.5	February 2002 to August 2012

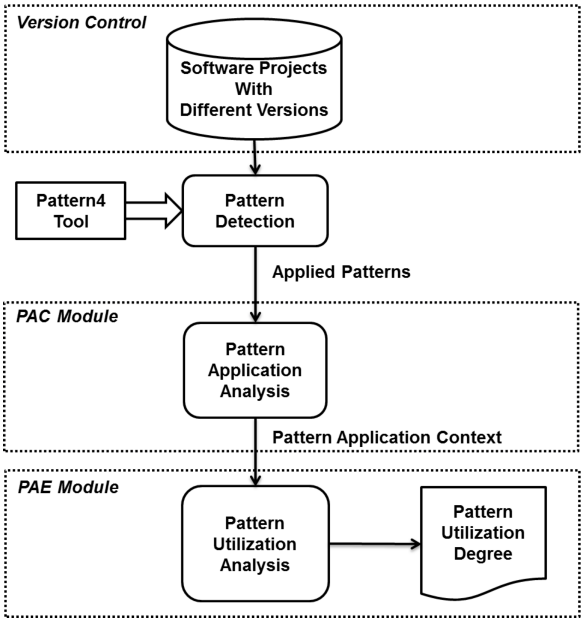

Fig. 7. System architecture of *Pattern Utilization Analyzer*.

steps are accompanied with a real case of JHotDraw open source project and *Observer* pattern.

Step 1: Collecting source code files. The first step is to collect source code files of JHotDraw software. We collect them from the SourceForge web site. Those files are then imported to our *PUA* tool and are managed by the version control module.

Step 2: Detecting deployed design patterns in the software. The second step is to detect the deployed patterns in each version of the collected OSS projects. The tool identifies design patterns by using a graph-matching based approach proposed by Tsantalis *etal*. [28], which is based on similarity scoring between graph vertexes. The tool can detect design patterns for: *Creational* design patterns (*Factory Method, Prototype*), *Structural* design patterns (*Adapter-Command, Composite, Decorator*) and *Behavioral* design patterns (*Observer, State-Strategy, Template method, Visitor*).

The detected results of each version are managed by the version control module of the *PUA*. The *PAC* module stores the defined context of each design pattern and the *EPU* module identifies whether the deployed pattern conforms the related *EPU*.

Step 3: Performing utilization analysis by the *PUA*. In this step, the utilization analysis of design patterns in the evolution are evaluated by the detected design patterns of each version of selected open source software. A design pattern is declared to be utilized when its *PAC* evolves conforms to the *EPU*. The *DoU* is evaluated by the *PAC* and *EPU* modules of *PUA*.

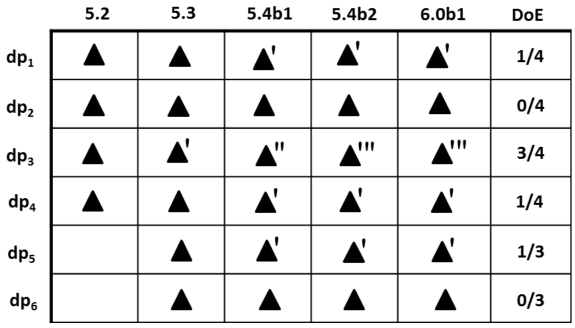

Fig. 8. The analysis results of *Observer* pattern for JHotDraw.

Step 4: Evaluating the *DoU* of design patterns in software evolution. In the final step, the analysis results are generated to show the degree of utilization of each deployed design pattern in the evolution.

In Fig. 8, there are six *Observer* design pattern instances (dp_1-dp_6) deployed for version 5.2, 5.3, 5.4b1, 5.4b2 and 6.0b1 of JHotDraw software. A design pattern dp_1 is deployed at version 5.2 and stays alive consecutively for five versions. The number of evolution is four and the number of utilized application is one for version 5.3 to 5.4b1. The *DoU* of dp_1 is 1/4.

Patterns dp_2, dp_3 and dp_4 are all deployed at version 5.2 and stays alive consecutively till version 6.0b1. The number of evolution is four for each pattern instance and the number of utilized applications are zero, three and one for dp_2, dp_3 and dp_4. The evaluated *DoU* of dp_2, dp_3 and dp_4 is 0/4, 3/4 and 1/4 respectively.

Pattern instances dp_5 and dp_6 are both deployed at version 5.3 and stays alive consecutively till version 6.0b1. The number of evolution is three for each pattern and the number of utilized applications is one and zero for dp_5 and dp_6. The *DoU* of dp_5 and dp_6 is 1/3 and 0/3, respectively.

4.3 Experimental Results

Table 3 shows the *DoU* experimental results for the fifteen open source projects. Abbreviations of Design Pattern column refers to the following: FM=*Factory Method*, P=*Prototype*, AC=*Adapter-Command*, C=*Composite*, D=*Decorator*, O=*Observer*, SS=*State-Strategy*, TM=*Template Method*, V=*Visitor*.

The experimental results show that most of the *DoU* results are universally not as high as expected in comparison with the maximum value 0.35. The blank field indicates the relative pattern is not deployed in the project and the zero value means the pattern is deployed but not utilized. Referring to Table 3, we can observe the design patterns are commonly utilized in the projects as Jena, JGnash, JHotDraw and JMemorize. The JHotDraw project performs well than the other three projects. The highest *DoU* among these projects and patterns is 0.350 of *Observer* pattern for the JHotDraw project.

For the FreeMind project, the evaluated results are relatively higher than all the other projects. *Prototype*, *Command* and *Visitor* patterns are not deployed in FreeMind project and *Decorator*, *Observer* and *State-Strategy* patterns are performed well as the *DoU* values are larger than 0.3. HtmlUnit is another project that the design patterns are commonly deployed in the project. *Composite* and *Visitor* patterns are not deployed in HtmlUnit project. The highest *DoU* value is 0.196 of *Decorator* pattern.

The *DoU* evaluation results are poorly performed in Jaxe, JConvert and JEdit. There are also few patterns utilized in these three projects. *Adapter-Command* pattern is utilized in Jaxe and JEdit projects. *State-Strategy* pattern pair is utilized in Jaxe and JConvert projects. *Template method* pattern is utilized in JConvert.

Composite pattern performs better than other patterns for LatexDraw project. *Factory Method* pattern plays an important role for TuxGuitar project. For Weka, the *State-Strategy* pattern pair is the highest utilized pattern pair. There are only three patterns deployed in PDFsam project but only *State-Strategy* pattern pair is utilized. The *DoU* results are relatively low or not applicable for JUnit and Pixelitor projects.

5. ANALYSIS AND DISCUSSION

The deployment of design patterns is expected to enhance the design flexibility to reduce the subsequent maintenance effort. But the experimental results reveal that the *DoU* results are not as good as expected. Further statistical analysis is conducted to investigate the following research questions.

5.1 Research Questions

RQ1: Is there any statistical difference between the *DoU* of deployed patterns and their type?

Programmers deploy design patterns and wish to reuse them in subsequent designs. The *DoU* should be higher for the deployed design patterns if they were correctly deployed and effectively utilized in the software evolution. This question is concerned with investigating whether there will be significant differences between the utilization of deployed design patterns and their type.

Table 3. *DoU* results of evaluated open source projects.

	FM	P	AC	C	D	O	SS	TM	V
FreeMind	0.161		0.165		0.333	0.333	0.301	0.263	
HtmlUnit	0.082	0	0.042		0.196	0.032	0.129	0.101	
Jaxe	0		0.007		0	0	0.014	0	
JConvert			0			0	0.158	0.125	
JEdit			0.091				0		
Jena	0.056	0.060	0.014	0.056	0.076	0.071	0.093	0.047	0
JGnash	0.012	0.001	0.023	0	0.023	0.017	0.041	0.017	
JHotDraw	0.143	0.004	0.044	0.063	0.162	0.350	0.176	0.104	0
JMemorize	0.010	0	0.020	0	0.010	0.040	0.030	0.030	
JUnit	0		0.016	0.045	0	0	0.120	0.026	
LatexDraw	0	0.005	0.058	0.261		0.174	0.073	0.075	
PDFsam			0				0.029	0	
Pixelitor	0.011	0	0.007			0.020	0.016	0.029	
TuxGuitar	0.200	0.009	0.062	0		0.125	0.051	0.032	
Weka	0.068	0	0.052		0.145	0	0.315	0.013	

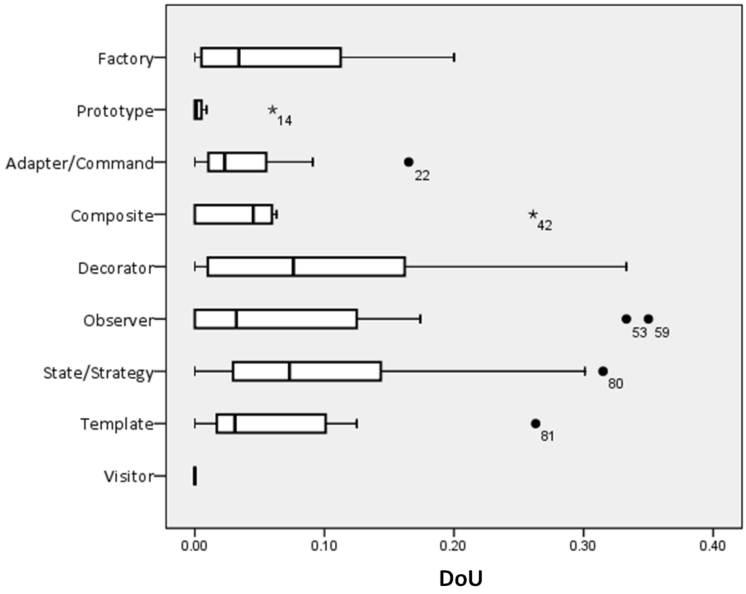

Fig. 9. Box plots of the *DoU* analysis results for different patterns.

RQ2: Is there any statistical difference between the *DoU* of deployed patterns and the open source project?

The design of a project may affect the choice of deployed design patterns according to different application domains. The *DoU* should be significantly different for some specific project evaluation. This question concerns whether there will be significant differences between the utilization of deployed design patterns and the design of open source projects.

5.2 Relationship Between *DoU* and Design Patterns

To answer RQ1, a one way analysis of variance is performed to analyze the relationship between *DoU* and design patterns. Table 4 presents the statistical results thereof. Fig. 9 graphically plots the results of the experiment. In this box plot, the x-axis represents the utilization on the graphical scale, and the y-axis represents the design pattern. To complement the graphical presentation, a number of statistical results were obtained. In all tests, a significance level of 0.05 was used.

Table 4. Statistics description: One Way ANOVA of *DoU* for Multiple design patterns.

	SS	df	SS/df	F-distribution	P-value
Between	0.090	8	0.011	1.569	0.146
Within	0.626	87	0.007		
Total	0.716	95			

Null Hypothesis All pairs of patterns will have the same means (μ) to the degree of utilization of deployed design patterns. H_0: $\mu_1 = \mu_2 = ... = \mu_i$ *(where i depicted a total of i*

different design patterns)

Alternate Hypothesis Each pair of patterns will have a difference mean (μ) degree of utilization from that of the deployed design patterns. H_1: $\mu_1 \neq \mu_2 \neq ... \neq \mu_i$ *(where i depicted a total of i different design patterns)*

Statistical Analysis ANOVA yielded the following results. Since (p-value= 0.146 $>$ 0.05 = α), the null hypothesis cannot be rejected, suggesting that the degree of utilization of deployed design patterns do not depend on the type of deployed design pattern. The *DoU* evaluation results do not have significant differences between different types of design patterns and the mean values are universally less than 0.2. It indicates that the utilization of design patterns could not meet the original expected design purposes. Possible reasons for the inference could be: (1) Although design patterns have widely varying complexities and applicability, the deployment of design patterns in software design is still affected by various factors, such as human habits and program comprehension [21]. (2) In software design, programmers are likely to choose the patterns with which they are familiar. Sometimes, the structural complexity influences the willingness of programmers to apply design patterns [30].

5.3 Relationship Between *DoU* and Open Source Projects

To answer RQ2, a one way analysis of variance method is used to analyze the relationship between *DoU* and design of open source projects. Table 5 presents the statistical results thus obtained. Fig. 10 graphically plots results of the experiment.

The finding of a statistically significant effect in ANOVA is commonly followed up with at least one other tests either to assess which groups differ from which other groups or to test various other hypotheses. Follow-up tests are distinguished by whether they are planned (*a priori*) or *post hoc*.

Table 5 presents the mean values of variables that were analyzed using an ANOVA. The variables are then assigned a letter, provided as a superscript, based on a Scheffé contrast (ref. Table 6). Values that vary significantly based on the *post-hoc* Scheffé contrast have different superscripts.

Table 5. Statistics description: One Way ANOVA of DoU for Multiple open source projects.

	SS	df	SS/df	F-distribution	P-value
Between	0.356	14	0.025	5.704	$<$0.001
Within	0.361	81	0.004		
Total	0.716	95			

Null Hypothesis All pairs of software projects have same mean (μ) degree of utilization of deployed design patterns. H_0: $\mu_1 = \mu_2 = ... = \mu_j$ *(where j depicted a total of j different software projects)*

Alternate Hypothesis Each pair of software projects' deployed design patterns will have different mean (μ) degrees of utilization. H_1: $\mu_1 \neq \mu_2 \neq ... \neq \mu_j$ *(where j depicted*

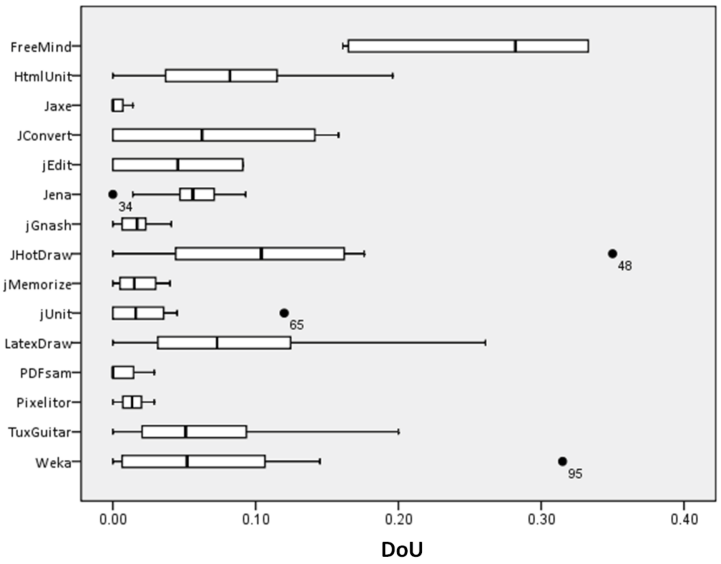

Fig. 10. Box plots of the *DoU* analysis results for different projects.

Table 6. The one-way analysis of variance (ANOVA) method with a post-hoc test.

Scheffé's method						
(I) OSS	(J) OSS	Group mean difference (I-J)	SD	P-value	95% confidence interval	
					Lower bound	Upper bound
FreeMind	Jaxe	0.255833*	0.038531	0.001	0.06156	0.4501
	Jena	0.206778*	0.035174	0.006	0.02943	0.38412
	jGnash	0.242583*	0.036042	<0.001	0.06086	0.42431
	jMemorize	0.241833*	0.036042	<0.001	0.06011	0.42356
	jUnit	0.229762*	0.037129	0.002	0.04256	0.41697
	PDFsam	0.249667*	0.047191	0.028	0.01173	0.4876
	Pixelitor	0.245500*	0.038531	0.001	0.05123	0.43977
	TuxGuitar	0.190905*	0.037129	0.04	0.0037	0.37811
* Group mean difference at P-value<0.05 level is significant						

a total of j different software projects)

Statistical Analysis The ANOVA-test for equality of means in the 15 software projects yields a very small value (p-value= $0.000 < 0.05 = \alpha$), so the hypothesis of equal mean scores of projects is rejected. Restated, the degrees of utilization of deployed design patterns varied significantly among different application domain software projects. Scheffé's method[1] subsequently applied contrasts projects in a non pairwise manner. The *post-hoc* test results reveal that the FreeMind software project differs significantly from another eight software projects.

According to the experimental results herein, in the JHotDraw project, the highest *DoU* is 0.350 for the *Observer* pattern. In the FreeMind project the highest *DoU* is 0.333

[1]Frequently, superscript letters are used to indicate which values are significantly different using the Scheffé method. For example, when mean values of variables that have been analyzed using an ANOVA are presented in a table, they are assigned a different letter superscript based on a Scheffé contrast.

for the *Observer* and *Decorator* patterns, indicating that the *Observer* pattern is crucial to MVC user interface design. In JHotDraw and FreeMind, *Observer* patterns are used to implement the notification-listening mechanism that manages the updating of figure visualization following changes [2]. Since the *DoU* values in FreeMind are commonly higher than often exceeds those in other projects, the deployment of design patterns in the FreeMind project is frequently examined and the design goals associated with the selected patterns are satisfied in the evolution of the software.

5.4 Threats to Validity

This section discusses threats to validity that can affect the results reported in this paper following a well-known template [25].

Threats to construct validity concern the relationship between theory and observation. They can be due to the evaluation performed, in particular related to design pattern identification. We are aware that our results can be influenced by the precision and recall, of the Tsantalis *etal*. tool [28]. However, in their work Tsantalis *etal*. [28] showed that for JHotDraw which is the only open source project where design patterns are well documented. The precision of design pattern identification is 100% and recall is 100%. To further inspect other researches which apply the tool to preform experiments. In the work of Aversano *etal*. [1], the tool precision inspection is overall above 85%. The tool limits the influence of false positives on our results.

Threats to internal validity can be due to the influence of external factors on the relationship object of the study, that is, the relationship between design pattern utilization degree and the projects and the types of design patterns. We analyzed the influence of two external factors, that is, the project and the kind of pattern, by means of a one-way ANOVA. The results indicate that the project factor has a significant influence and that the pattern factor has no significant influence.

Threats to external validity are the degrees to which the results are generalizable. We select fifteen open source software systems from different domains and different sizes. Nevertheless, it would be desirable to analyze further systems to draw more general conclusions. Due to the research purpose, we consider a subset of 9 out of the 23 patterns from the Gamma *etal*. catalogue which emphasize on the defined *quality-improver* patterns in our previous work [14].

Regarding reliability validity, the evaluated source code of the fifteen software systems and the design pattern detection tool is publicly available. The experiment procedure for the analysis is described in detail in Section 4.2, and we make raw data available to allow for replicating statistical analysis.

6. CONCLUSION

Software is constantly evolving over time because it must meet changing requirements. The deployment of design patterns provides a flexible architecture that makes the software system reusable and extensible. Although applying design patterns can improve system quality, the utilization of their deployment must be examined.

As we mentioned, most patterns address the maintainability issues, the evaluation should be base on all versions in its software evolution. In our study, we propose a methodology to classify design patterns and analyze their utilization. We also develop an evolutionary analysis method to observe the utilization of deployed design patterns between consecutive versions in the evolution instead of a single version snapshot. Experi-

ments are performed on fifteen open source software to realize the utilization of deployed design patterns during their software evolution.

A t-test followed by ANOVA method is used to better judge if the *DoU* effect of the deployed design patterns is really significant in the software evolution. The analysis results show that the utilization of deployed design patterns does not have significant difference among the design patterns in the evolution of software design. Most of the deployed design patterns are not utilized against the original design, such that design effort is less utilized. Many of these deployed design patterns are not even reused after their initial deployment.

The innovative approach presented herein is to assess whether deployed design patterns can satisfy the original design requirements. Our proposed evaluation can

- help developers know the status of deployed design pattern applications during the project evolution;
- assist analysts to assess the status of deployed design patterns for enhancing software maintenance actions, for example, refactoring to utilized pattern, or to remove design pattern;
- suggest managers where to locate the problems in function planning if the requirement changes do not meet the original design purpose.

In future work, we plan to implement the tool as a cloud service to improve the computation performance. We also plan to add more types of design patterns to enhance the capability of design pattern detection. With an improved tool, we will have efficient evaluation tool and perform experiments on more open source software systems.

REFERENCES

1. L. Aversano, L. Cerulo, and M. Di Penta, "Relationship between design patterns defects and crosscutting concern scattering degree: an empirical study," *IET Software*, Vol. 3, 2009, pp. 395-409.
2. Lerina Aversano, Gerardo Canfora, Luigi Cerulo, Concettina Del Grosso, and Massimiliano Di Penta, "An empirical study on the evolution of design patterns," in *Proceedings of the 6th Joint Meeting of European Software Engineering Conference and ACM SIGSOFT Symposium on Foundations of Software Engineering*, 2007, pp. 385-394.
3. B. Boehm and H. In, "Identifying quality-requirement conflicts," *IEEE Software*, Vol. 13, 1996, pp. 25-35.
4. L. Chung, K. Cooper, and A. Yi, "Developing adaptable software architectures using design patterns: an nfr approach," *Computer Standards & Interfaces*, Vol. 25, 2003, pp. 253-260.
5. J. Dong, S. Yang, and K. Zhang, "Visualizing design patterns in their applications and compositions," *IEEE Transactions on Software Engineering*, Vol. 33, 2007, pp. 433-453.
6. J. Dong, D. S. Lad, and Y. Zhao, "Dp-miner: Design pattern discovery using matrix," in *Proceedings of the 14th Annual IEEE International Conference and Workshops on Engineering of Computer-Based Systems*, 2007, pp. 371-380.
7. C. Ebert, "Putting requirement management into praxis: dealing with nonfunctional requirements," *Information and Software Technology*, Vol. 40, 1998, pp. 175-185.

8. M. Fowler, "Patterns," *IEEE Software*, Vol. 20, 2003, pp. 56-57.
9. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Software*, Addison-Wesley, MA, 1994.
10. A. R. Graves and C. Czarnecki, "Design patterns for behavior-based robotics," *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, Vol. 30, 2000, pp. 36-41.
11. D. Gross and E. Yu, "From non-functional requirements to design through patterns," *Requirements Engineering*, Vol. 6, 2001, pp. 18-36.
12. N. L. Hsueh, L. C. Wen, D. H. Ting, W. Chu, C. H. Chang, and C. S. Koong, "An approach for evaluating the effectiveness of design patterns in software evolution," in *Proceedings of IEEE 35th Annual Computer Software and Applications Conference*, 2011, pp. 315-320.
13. N. L. Hsueh, P. H. Chu, and W. Chu, "A quantitative approach for evaluating the quality of design patterns," *Journal of Systems and Software*, Vol. 81, 2008, pp. 1430-1439.
14. N. L. Hsueh, J. Y. Kuo, and C. C. Lin, "Object-oriented design: A goal-driven and pattern-based approach," *Software and Systems Modeling*, Vol. 8, 2009, pp. 67-84.
15. Brian Huston, "The effects of design pattern application on metric scores," *Journal of Systems and Software*, Vol. 58, 2001, pp. 261-269.
16. C. Izurieta and J. M. Bieman, "A multiple case study of design pattern decay, grime, and rot in evolving software systems," *Software Quality Journal*, Vol. 21, 2013, pp. 289-323.
17. E. Yu L. Chung, B.A. Nixon, and J. Mylopoulos, *Non-Functional Requirements in Software Engineering*, Kluwer Academic Publishing, NY, 2000.
18. J. Lee and J. Y. Kuo, "New approach to requirements trade-off analysis for complex systems," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 10, 1998, pp. 551-562.
19. A. D. Lucia, V. Deufemia, C. Gravino, and M. Risi, "Design pattern recovery through visual language parsing and source code analysis," *Journal of Systems and Software*, Vol. 82, 2009, pp. 1177-1193.
20. T. H. Ng, S. C. Cheung, W. K. Chan, and Y. T. Yu, "Do maintainers utilize deployed design patterns effectively?" in *Proceedings of IEEE 29th International Conference on Software Engineering*, 2007, pp. 168-177.
21. T. H. Ng, Yuen Tak Yu, S. C. Cheung, and W. K. Chan, "Human and program factors affecting the maintenance of programs with deployed design patterns," *Information and Software Technology*, Vol. 54, 2012, pp. 99-118.
22. N. Pettersson, W. Lö andwe, and J. Nivre, "Evaluation of accuracy in design pattern occurrence detection," *IEEE Transactions on Software Engineering*, Vol. 36, 2010, pp. 575-590.
23. L. Prechelt, B. Unger, W. F. Tichy, P. Brossler, and L. G. Votta, "A controlled experiment in maintenance: comparing design patterns to simpler solutions," *IEEE Transactions on Software Engineering*, Vol. 27, 2001, pp. 1134-1144.
24. L. Prechelt, B. Unger-Lamprecht, M. Philippsen, and W. F. Tichy, "Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance," *IEEE Transactions on Software Engineering*, Vol. 28, 2002, pp. 595-606.
25. R. K. Yin, *Case Study Research: Design and Methods*, SAGE Puclications, NY, 2002.
26. D. C. Schmidt, "Using design patterns to develop reusable object-oriented communication software," *Communications of the ACM*, Vol. 38, 1995, pp. 65-74.

27. L. Tahvildari and K. Kontogiannis, "A software transformation framework for quality-driven object-oriented re-engineering," in *Proceedings of IEEE International Conference on Software Maintenance*, 2002, pp. 596-605.
28. N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S.T. Halkidis, "Design pattern detection using similarity scoring," *IEEE Transactions on Software Engineering*, Vol. 32, 2006, pp. 896-909.
29. N. Tsantalis, "Design pattern detection using similarity scoring," http://java.uom.-gr/nikos/pattern-detection.html, 2011.
30. M. Vokáč, "Defect frequency and design patterns: an empirical study of industrial code," *IEEE Transactions on Software Engineering*, Vol. 30, 2004, pp. 904-917.
31. T. Winn and P. Calder, "Is this a pattern?" *Software, IEEE*, Vol. 19, 2002, pp. 59-66.
32. Cheng Zhang and David Budgen, "What do we know about the effectiveness of software design patterns?" *IEEE Transactions on Software Engineering*, 2012, Vol. 38, pp. 1213-1231.
33. C. Zhang and D. Budgen, "A survey of experienced user perceptions about software design patterns," *Information and Software Technology*, Vol. 55, 2013, pp. 822-835.
34. J. Zhu and P. Jossman, "Application of design patterns for object-oriented modeling of power systems," *IEEE Transactions on Power Systems*, Vol. 14, 1999, pp. 532-537.

Nien-Lin Hsueh is a Professor in the Department of Information Engineering and Computer Science at Feng Chia University in Taiwan, and the Director of the Department of Information Engineering and Computer Science at FCU since 2018. Prof. Hsueh is currently the Chairman of of Software Engineering Association Taiwan (SEAT; https://www.seat.org.tw) since 2017. His research interests include software engineering, object-oriented methodologies, software process improvement, and e-learning. He was the technology consultant in the CMMI-based process improvement project of Office of Information Technology at Feng Chia University. He received his Ph.D. in Computer Science from National Central University in Taiwan.