Abstract
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.
Design Pattern Analysis with Software Evolution Data

NIE-N-LIN HSUEH
Department of Information Engineering and Computer Science
Feng Chia University
No. 100, Road Wen-Hwa, Taichung City, 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

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, 27]. A growing number of practitioners have shown great interests in using design patterns towards high-quality software [4, 10, 11, 44, 51].

Although design patterns have been widely applied, its contribution is questioned and various approaches are proposed to validate the correctness of design pattern application [22, 24, 25, 49] and its value in quality improvement [1, 13, 50]. To achieve that, most of these researches analyze the programs with a given quality model [2, 13, 15, 47]. 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 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 [42]. 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 researches. In section 2.2, we introduce the two pattern detection work since they are important background techniques for the pattern evaluation researches.

![Fig. 1. Less utilized design pattern application in software evolution.](image)
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 et al. [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. [24] test whether the use of some specific design patterns is helpful for participants with different backgrounds. Prechelt et al. [25] 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. [22] 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. Aversano et al. [1] report a correlation between crosscutting concern scattering degree and the number of defects in design pattern classes. Vokáč [47] 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. [21] 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. [49] 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. [50] 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 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.

<table>
<thead>
<tr>
<th>Research</th>
<th>Motivation</th>
<th>Method</th>
<th>Code Analysis</th>
<th>Subjects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Huston [15]</td>
<td>Analyze the compatibility between design patterns and design metrics</td>
<td>Analysis</td>
<td>Yes</td>
<td>Design Patterns</td>
</tr>
<tr>
<td>Hsueh [13]</td>
<td>Validate if a design pattern well-designed</td>
<td>Quantitative Method</td>
<td>No</td>
<td>Design Patterns</td>
</tr>
<tr>
<td>Izurieta and Bierman [16]</td>
<td>Examine the extent to which software designs decay, rot, and accumulate grime</td>
<td>Tools</td>
<td>Yes</td>
<td>Software</td>
</tr>
<tr>
<td>Zhang and Budgen [49]</td>
<td>Investigate how extensively the use of design patterns and how and when they can provide effective mechanism for knowledge transfer</td>
<td>Survey</td>
<td>No</td>
<td>Research Papers</td>
</tr>
<tr>
<td>Zhang and Budgen [50]</td>
<td>Identify which design patterns are considered to be useful by experienced users</td>
<td>Survey</td>
<td>No</td>
<td>Human</td>
</tr>
<tr>
<td>Prechelt et al. [24]</td>
<td>Investigate whether using design patterns is beneficial even the actual design is simpler</td>
<td>Controlled Experiments</td>
<td>No</td>
<td>Human</td>
</tr>
<tr>
<td>Prechelt et al. [25]</td>
<td>Test if the design patterns in the program code are documented explicitly can help the maintainer</td>
<td>Controlled Experiments</td>
<td>No</td>
<td>Human</td>
</tr>
<tr>
<td>Aversano et al. [2]</td>
<td>Analyze how frequently patterns are modified, to what changes undergo and what classes co-change with the patterns</td>
<td>Tools</td>
<td>Yes</td>
<td>Software</td>
</tr>
<tr>
<td>Aversano et al. [1]</td>
<td>Investigate whether the defects in design pattern’s code is correlated to the scattering degree of their induced crosscutting concerns</td>
<td>Tools</td>
<td>Yes</td>
<td>Software</td>
</tr>
<tr>
<td>Vokáč [47]</td>
<td>Investigate if design with properly applied patterns reduces defect frequency</td>
<td>Self-developed Tool</td>
<td>Yes</td>
<td>Software</td>
</tr>
<tr>
<td>Ng et al. [21]</td>
<td>Analyze how maintainers utilize the design patterns to complete anticipated changes in software</td>
<td>Experiments</td>
<td>Yes</td>
<td>Software</td>
</tr>
<tr>
<td>Ng et al. [22]</td>
<td>Identify factors with contributions to the maintenance of programs with deployed design patterns</td>
<td>Experiments</td>
<td>No</td>
<td>Human</td>
</tr>
<tr>
<td>Hsueh [12]</td>
<td>Evaluate the effectiveness of deployed design pattern in software evolution</td>
<td>Self-developed Tool</td>
<td>Yes</td>
<td>Software</td>
</tr>
</tbody>
</table>

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 et al. [45] to identify the deployed design patterns for evaluation. Tsantalis et al. [45] use similarity scoring algorithm between graph representations to automatically detect modified 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 et al. [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 [20] or design [5]. Some works [6, 45] 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 perspectives 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 analysis, 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 analysis 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 requirements, 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.

- **Decomposition** activity is performed to decompose a system into simpler parts called subsystems for reducing the complexity of a system. The Observer pattern can help resolve problem related to decomposition by reducing dependencies between entity objects and boundary objects.

- **Object allocation** activity is performed to distribute objects or subsystems in different computers carefully to satisfy high performance needs or interconnecting multiple distributed users. The remote proxy pattern can be applied to enhance design simplicity by providing an agent object to hide the fact that an object resides in a different address.

- **Access control** activity is performed to define the access control policy in the stage of providing a more secure multi-user environment. Protection proxy design pattern plays a housekeeping role in the Proxy design pattern to filter inappropriate access.
• **Control flow** is performed to model the multiple simultaneous flows of control in a concurrent system to enhance the overall performance. *Command* design pattern centralizes a control flow into an object instead of spreading it between boundary and entity objects.

• **Component composition** is performed to select, adopt and compose commercial off-the-shelf components with the developing system or reducing the cost to confront a recurring problem. *Adapter* design pattern can be used to convert interfaces and serves as glue between these components and a developing system.

**Quality-improver.** Non-functional requirements are not easy to handle because they are subjective, relative and interacting [7, 18]. 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 [48]. 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. For example, considering the original intent described in *Observer* design pattern [9]:

> Define an one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

By elaborating the intent, we can understand the design pattern is designed to resolve the communication between a subject’s objects and its related observer objects. Viewed from the functional aspect, it requires the subject to notify all observers when it changes its state. Viewed from the non-functional aspect, it requires the notification to work automatically without knowing types of observers. Analyzing design pattern from requirement view is performed based on this observation to explore how a design pattern can improve non-functional requirements. This kind of patterns are the ones we address in our study.

**Conflict-resolver.** Finding the right balance for conflicting requirements is important in achieving successful software requirements and products [3, 19]. 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.

Fig. 4 illustrates an example of using design patterns for resolving conflicts. Both models (a) and (b) are designed to add responsibilities to a *Basic* object. In model (a), class inheritance mechanism is used to extend responsibilities transparently, that is, the *Basic* object does not have to be aware of the extension objects. Such a design may suffer the disadvantage of static inheritance: a large number of independent extensions would produce an explosion of subclasses to support every combination. On the other hand, model (b) utilizes an object composition approach to extend responsibilities in a flexible way. The responsibility can be attached to the *Basic* object at run-time dynamically. Furthermore, the combination of independent extensions would not produce an explosion of subclasses. However, such a design will lose the advantages of class inheritance. The *Basic* object must be aware of the expected extensions; this would increase the design
complexity of the Basic object. In a nutshell, model (b) provides a scalable design for adding new responsibilities but makes the basic object more complex; model (a) provides a simple design on the basic object, but is not easy to scale up. Decorator design pattern can be used to provide a compromise in this conflict by providing a mix structure of class inheritance and object composition [see model (c)].

![Diagram](image)

**Fig. 4. Resolving conflicts by Decorator design pattern.**

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.

A design pattern can be analyzed from different perspectives. For example, Observer design pattern can play the role of quality-improver to reduce the coupling, the role of activity-facilitator to facilitate design decomposition, as well as the role of problem solver to prevent data inconsistency. A design pattern with multiple roles can be applied in different phases and for resolving different problems.

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. 5 and 6 illustrate the FR-structure and its corresponding NFR-structure of Strategy design pattern. In Fig. 6, 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. 5. FR-structure of Strategy design pattern.](image1)

![Fig. 6. NFR-structure of Strategy design pattern.](image2)

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, classified as quality-improver, and their corresponding FR-intent and NFR-intent are listed in Table 2.

### 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.
Table 2. Expected utilization of design patterns.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Functional Intent</th>
<th>Non-Functional Intent</th>
<th>Expected Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Observer</td>
<td>A subject object can notify all related objects (called observers) when it changes state</td>
<td>A subject object can notify all observers automatically without knowing types of observers, thus reducing the coupling between the subject and observers</td>
<td>Add or remove ConcreteObserver class</td>
</tr>
<tr>
<td>Factory Method</td>
<td>Define an interface for creating an object and lets a class defer instantiation to subclasses</td>
<td>Easier to extend a family of products, thus reducing the coupling between concrete classes and the application</td>
<td>Add or remove ConcreteProduct class and ConcreteCreator class</td>
</tr>
<tr>
<td>Prototype</td>
<td>Define an abstract base class with a virtual clone method to maintain cloneable concrete derived classes</td>
<td>Eliminate the overhead of writing code that invokes the &quot;new&quot; operator</td>
<td>Add or remove ConcretePrototype type class</td>
</tr>
<tr>
<td>Adapter</td>
<td>Let classes with incompatible interfaces to work together</td>
<td>Work as a bridge to enhance extensibility; wrap an existing class with a new interface to improve reusability</td>
<td>Add or remove ConcreteAdapter class</td>
</tr>
<tr>
<td>Command</td>
<td>An object sends a request to another object</td>
<td>Easier to manipulate and extend the request by de-coupling the object that carries out the request from the one that knows how to perform it, thus enhancing extensibility</td>
<td>Add or remove Receiver or related ConcreteCommand class</td>
</tr>
<tr>
<td>Composite</td>
<td>To compose objects into tree structures to represent part-whole hierarchies and treat individual objects and compositions of objects uniformly</td>
<td>Easier to extend new kinds of components without knowing they are leaves or composite components</td>
<td>Add or remove Composite classes and leaf classes</td>
</tr>
<tr>
<td>Decorator</td>
<td>To provide a flexible alternative to subclassing for extending functionality</td>
<td>It is flexible than inheritance because of adding responsibility at run time and extends functionality of object without affecting any other object</td>
<td>Add or remove ConcreteDecorator class</td>
</tr>
<tr>
<td>State</td>
<td>Define a base class to present a single interface to encapsulate state-specific behavior in the appropriate State derived classes</td>
<td>Easier to replace the state-specific behavior with a new one, thus enhancing extensibility; the state base class is factorized in the super-class, thus improve reusability</td>
<td>Add or remove ConcreteState class</td>
</tr>
<tr>
<td>Strategy</td>
<td>An object uses an algorithm to resolve a specific problem</td>
<td>Easier to replace the algorithm with a new one, thus enhancing extensibility; the generic algorithm is factorized in the super-class, thus improve reusability</td>
<td>Add ConcreteStrategy class with new algorithm; remove ConcreteStrategy class with unnecessary algorithm</td>
</tr>
<tr>
<td>Template method</td>
<td>An object uses an algorithm to resolve a specific problem</td>
<td>Easier to redefine certain steps of an algorithm with changing the algorithm's structure, thus enhancing extensibility</td>
<td>Add ConcreteClass class with overriding template method or remove unnecessary ConcreteClass class</td>
</tr>
<tr>
<td>Visitor</td>
<td>Present an operator to be performed on the elements of an object structure</td>
<td>Easier to define a new operator without changing the classes of the elements on which it operates, thus enhancing extensibility</td>
<td>Add or remove ConcreteVisitor and related ConcreteElement classes</td>
</tr>
</tbody>
</table>

Table 2 lists quality-improver design patterns, their corresponding FR-intent and NFR-intent, and their expected utilization. Therefore, the non-functional intents and corresponding expected utilization of design patterns are the criteria to analyze the utilization of design patterns in software evolution. Referring to Table 2, we can inspect the quality-improver patterns are utilized if they meet the expected utilization criterion.

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:

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

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. Fig. 7 shows the $Pattern Application Context$ of a $Strategy$ pattern applied in a design $D$. The $PAC$ of $Strategy$ pattern in a design $D$ can be defined as:

$$PAC(Strategy, D) = \{ <DataCompression, RC>, <CompressionTool, RS>, <RAR, RCS>, <ZIP, RCS> \}, \quad (2)$$

where $RC$, $RS$, $RCS$ 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:

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

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:

$$EPU(Strategy) = \{ <RCS, +>, <RCS, −> \}. \quad (4)$$
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 \( utilized(dp, D_{j+1}) \) to represent if a design pattern \( dp \) is utilized at certain version \( D_{j+1} \). The formal definition is as below:

\[
utilized(dp, D_{j+1}) \iff \exists D_j \Delta(PAC(dp, D_j), PAC(dp, D_{j+1})) \text{ conforms to } EPU(dp),
\]

where \( \Delta(PAC(dp, D_j), PAC(dp, D_{j+1})) \) denotes the difference between \( PAC(dp, D_j) \) and \( PAC(dp, D_{j+1}) \). The design \( D_j \) and \( D_{j+1} \) deploy the same design pattern \( dp \). As presented in Fig. 8, the design \( D_j \) deploys a Strategy pattern. The intent behind the application of 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 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:

\[
DoU = \frac{\text{# of utilized applications}}{\text{# of pattern evolution}}
\]

The \( DoU \) can help us understand the utilized status of design pattern application. Take Fig. 9 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
Fig. 9. Utilization analysis of five instances during eight versions.

<table>
<thead>
<tr>
<th>Pattern instance</th>
<th>V1</th>
<th>V2</th>
<th>V3</th>
<th>V4</th>
<th>V5</th>
<th>V6</th>
<th>V7</th>
<th>V8</th>
<th>DoU</th>
</tr>
</thead>
<tbody>
<tr>
<td>dp1</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>2/7</td>
</tr>
<tr>
<td>dp2</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>3/7</td>
</tr>
<tr>
<td>dp3</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>1/5</td>
</tr>
<tr>
<td>dp4</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>X</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>1/2</td>
</tr>
<tr>
<td>dp5</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>▲</td>
<td>0/5</td>
</tr>
</tbody>
</table>

The number of evolution is seven and the number of utilized applications is three for \((V_3-V_4), (V_4-V_5)\) and \((V_5-V_7)\). The DoU of \(dp_2\) is 3/7. Pattern instance \(dp_3\) is deployed at \(V_5\) 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.

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 3 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 [35]. 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 [17]. JHotDraw is a popular evaluated software in many researches [45, 1, 23, 22]. 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 [45, 23]. JUnit is a simple framework for writing and running automated tests. As a political gesture, it celebrates programmers testing their own software [29]. Like JHotDraw as a graphic tool, FreeMind
is a premier free mind-mapping software which is written in Java language [37].

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 [32]. Jaxe is a free Java XML editor with a configurable GUI, using XML schemas for validation and XSL for exports in HTML or XML [33]. JConvert is a free unit conversion program that has a friendly user interface and can also be used with external applications [38].

JEdit is a multi-platform mature programmer’s text editor which is written in Java [28]. 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 [34]. JGnash is a cross platform personal finance application written in Java [39]. 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 [36].

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 [40]. PDFsam (PDF Split and Merge) is an easy to use tool to select files and set options [30]. Pixelitor is a free and open source image editing software that supports layers, image effects, multiple undo [31]. TuxGuitar is a multitrack guitar tablature editor and player written in Java-SWT. It can open GuitarPro, PowerTab and TablEdit files [43]. 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

<table>
<thead>
<tr>
<th>Project Name</th>
<th># of Evolution</th>
<th>Versions</th>
<th>Time Interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>FreeMind</td>
<td>16</td>
<td>0.0.2-0.9.0</td>
<td>June 2000 to February 2011</td>
</tr>
<tr>
<td>HtmlUnit</td>
<td>28</td>
<td>1.0.0-2.9.0</td>
<td>May 2002 to August 2011</td>
</tr>
<tr>
<td>Jaxe</td>
<td>38</td>
<td>1.0-3.5</td>
<td>June 2002 to June 2011</td>
</tr>
<tr>
<td>JConvert</td>
<td>9</td>
<td>1.0.0-1.1.0</td>
<td>August 2007 to May 2011</td>
</tr>
<tr>
<td>JEdit</td>
<td>27</td>
<td>2.3-4.5.1</td>
<td>January 2000 to March 2012</td>
</tr>
<tr>
<td>Jena</td>
<td>17</td>
<td>2.0-2.6.4</td>
<td>August 2003 to December 2010</td>
</tr>
<tr>
<td>JGnash</td>
<td>26</td>
<td>1.0.1-1.10.5</td>
<td>January 2003 to March 2006</td>
</tr>
<tr>
<td>JHotDraw</td>
<td>8</td>
<td>5.2-7.6</td>
<td>February 2001 to January 2011</td>
</tr>
<tr>
<td>JMemorize</td>
<td>12</td>
<td>0.7.0-1.3.0</td>
<td>October 2004 to March 2008</td>
</tr>
<tr>
<td>JUnit</td>
<td>19</td>
<td>3.4.0-4.10.0</td>
<td>December 2000 to September 2011</td>
</tr>
<tr>
<td>LatexDraw</td>
<td>24</td>
<td>1.0.2-2.0.8</td>
<td>January 2006 to March 2010</td>
</tr>
<tr>
<td>PDFsam</td>
<td>14</td>
<td>1.0.0-2.2.1</td>
<td>August 2007 to July 2012</td>
</tr>
<tr>
<td>Pixelitor</td>
<td>34</td>
<td>0.1.0-1.1.2</td>
<td>November 2009 to November 2010</td>
</tr>
<tr>
<td>TuxGuitar</td>
<td>11</td>
<td>0.2-1.2</td>
<td>June 2006 to November 2009</td>
</tr>
<tr>
<td>Weka</td>
<td>13</td>
<td>3.0.1-3.7.5</td>
<td>February 2002 to August 2012</td>
</tr>
</tbody>
</table>
directly to a dataset or called from your own Java code [41].

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. 10 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 [46], 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 steps are accompanied with a real case of JHotDraw open source project and Observer pattern.

![System Architecture Diagram](image.png)

**Fig. 10. System architecture of Pattern Utilization Analyzer.**

**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 [35]. 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 et al. [45], 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.

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. 11, there are six Observer design pattern instances (dp1 - dp6) deployed for version 5.2, 5.3, 5.4b1, 5.4b2 and 6.0b1 of JHotDraw software. A design pattern dp1 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 dp1 is 1/4.

Patterns dp2, dp3 and dp4 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 dp2, dp3 and dp4. The evaluated DoU of dp2, dp3 and dp4 is 0/4, 3/4 and 1/4 respectively.

Pattern instances dp5 and dp6 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 dp5 and dp6. The DoU of dp5 and dp6 is 1/3 and 0/3, respectively.

![Fig. 11. The analysis results of Observer pattern for JHotDraw.](image-url)
4.3 Experimental Results

Table 4 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 4, 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-

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

Table 4. DoU results of evaluated open source projects.
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.

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 5 presents the statistical results thereof. Fig. 12 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 5. Statistics description: One Way ANOVA of DoU for Multiple design patterns.

<table>
<thead>
<tr>
<th></th>
<th>SS</th>
<th>df</th>
<th>SS/df</th>
<th>F-distribution</th>
<th>P-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Between</td>
<td>0.090</td>
<td>8</td>
<td>0.011</td>
<td>1.569</td>
<td>0.146</td>
</tr>
<tr>
<td>Within</td>
<td>0.626</td>
<td>87</td>
<td>0.007</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>0.716</td>
<td>95</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Null Hypothesis

All pairs of patterns will have the same means ($\mu$) 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
Alternate Hypothesis

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

Statistical Analysis

ANOVA yielded the following results. Since ($p$-value $= 0.146 > 0.05 = \alpha$), 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:

- 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 [22].
- 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 [47].
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 6 presents the statistical results thus obtained. Fig. 13 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 6 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 7). Values that vary significantly based on the post-hoc Scheffé contrast have different superscripts.


<table>
<thead>
<tr>
<th></th>
<th>SS</th>
<th>df</th>
<th>SS/df</th>
<th>F-distribution</th>
<th>P-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Between</td>
<td>0.356</td>
<td>14</td>
<td>0.025</td>
<td>5.704</td>
<td>&lt;0.001</td>
</tr>
<tr>
<td>Within</td>
<td>0.361</td>
<td>81</td>
<td>0.004</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>0.716</td>
<td>95</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 7. The one-way analysis of variance (ANOVA) method with a post-hoc test.

<table>
<thead>
<tr>
<th>(I) OSS</th>
<th>(J) OSS</th>
<th>Group mean difference (I-J)</th>
<th>SD</th>
<th>P-value</th>
<th>95% confidence interval</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Lower bound</td>
<td>Upper bound</td>
<td></td>
</tr>
<tr>
<td>Jaxe</td>
<td>FreeMind</td>
<td>0.255833*</td>
<td>0.038531</td>
<td>0.001</td>
<td>0.06156</td>
</tr>
<tr>
<td>Jena</td>
<td>FreeMind</td>
<td>0.206778*</td>
<td>0.035174</td>
<td>0.006</td>
<td>0.02943</td>
</tr>
<tr>
<td>jGnash</td>
<td>FreeMind</td>
<td>0.242583*</td>
<td>0.036042</td>
<td>&lt;0.001</td>
<td>0.06086</td>
</tr>
<tr>
<td>jMemorize</td>
<td>FreeMind</td>
<td>0.241833*</td>
<td>0.036042</td>
<td>&lt;0.001</td>
<td>0.06011</td>
</tr>
<tr>
<td>jUnit</td>
<td>FreeMind</td>
<td>0.229762*</td>
<td>0.037129</td>
<td>0.002</td>
<td>0.04256</td>
</tr>
<tr>
<td>PDFsam</td>
<td>FreeMind</td>
<td>0.249667*</td>
<td>0.047191</td>
<td>0.028</td>
<td>0.01173</td>
</tr>
<tr>
<td>Pixelitor</td>
<td>FreeMind</td>
<td>0.245500*</td>
<td>0.038531</td>
<td>0.001</td>
<td>0.05123</td>
</tr>
<tr>
<td>TuxGuitar</td>
<td>FreeMind</td>
<td>0.190905*</td>
<td>0.037129</td>
<td>0.04</td>
<td>0.0037</td>
</tr>
</tbody>
</table>

*: Group mean difference at P-value <0.05 level is significant

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 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 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 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 [26].

\footnote{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.}
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 et al. tool [45]. However, in their work Tsantalis et al. [45] 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 et al. [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 et al. 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 based 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. Experiments 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.

ACKNOWLEDGMENT

This research was supported by the National Science Council, Taiwan R.O.C., under grants MOST 106-2221-E-035-MY2.

REFERENCES

9. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Software. Addison-Wesley, 1994.

**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 Honor Program, College of Information and Electrical Engineering at FCU since 2016. Prof. Hsueh is currently the Chairman 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.