Impact of Design Pattern Implementation Variants on the Retrieval Effectiveness of a Recovery Tool: An Exploratory Study

Conference Paper · August 2018

DOI: 10.1109/SEAA.2018.00034

CITATIONS
0

READS
34

4 authors, including:

Vincenzo Deufemia
Università degli Studi di Salerno
111 PUBLICATIONS **838** CITATIONS

SEE PROFILE

Carmine Gravino
Università degli Studi di Salerno
141 PUBLICATIONS **1,466** CITATIONS

SEE PROFILE

Michele Risi
Università degli Studi di Salerno
100 PUBLICATIONS **736** CITATIONS

SEE PROFILE

Impact of Design Pattern Implementation Variants on the Retrieval Effectiveness of a Recovery Tool: An Exploratory Study

Andrea De Lucia, Vincenzo Deufemia, Carmine Gravino, Michele Risi
Department of Computer Science
University of Salerno
Fisciano(SA), Italy
{adelucia, deufemia, gravino, mrisi}@unisa.it

***Abstract*—This paper investigates how *implementation variants* of design patterns impact on the retrieval effectiveness of a design pattern recovery tool. Specifically, we first defined several implementation variants of Adapter and Observer design patterns, by introducing constraints or relaxations on their canonical form. Then, we analyze the relationship between the complexity of these definitions and the precision and time needed by a design pattern recovery process we proposed in the past. To this end, we apply ePAD, an Eclipse plug-in for design pattern recovery, to eight software systems. We show that there exist interesting issues about the relationship between the complexity of the defined variants and the precision and time needed to recover their instances.**

Keywords- Design Pattern Recovery; Program Analysis; Implementation Variants of Patterns

I. INTRODUCTION

One of the main challenges in the recognition of design pattern instances in source code derives from the abstract and informal definition of design patterns [3][9]. Indeed, the definition of a design pattern comes with a general form (the two most popular are called *GoF form* and *canonical form*), consisting of several sections describing the design structure, participants, collaborations and so on. An intrinsic property of such design pattern definitions is that they can be implemented very differently depending on the programming language, the personal style, design decisions, and so on [7].

Recently, many efforts have been devoted to the development of tools that automate design pattern recovery from source code [1][3][17][20]. Some of them consider a single implementation variant for each pattern, e.g. [10], others aim at detecting variants, e.g. [16]. The presence of different variants in software systems and the ability of detection tools in recovering them make the comparison of design pattern detection tools a challenging activity [13].

Despite many proposals for recovering design pattern instances and their variants, no previous work has empirically analyzed how developers implement design pattern variants, and their impact on the performances of design pattern recovery strategies.

In this paper, we present a preliminary study aimed at analyzing how design pattern implementation variants impact on the retrieval effectiveness of a design pattern recovery tool. To this end, we first define several implementation variants by introducing constraints and relaxations into canonical definitions provided by Gamma *et al.* [7] and highlighting their effect on the recovery process as well as the asymptotic complexity variation. We configure ePAD [5], an Eclipse plug-in for design pattern recovery, to work with the defined implementation variants. Then, we present an exploratory study investigating the relations between the implementation variants of Adapter and Observer design patterns recovered from eight open source software systems and the performances of the recovery process in terms of precision and time. The Adapter pattern is commonly used to make existing classes work with others without modifying their source code, whereas the Observer pattern is manly used to implement distributed event handling systems in event-driven software [7].

The paper is structured as follows. Section II reports on design pattern recovery approaches that deal with the retrieval of design pattern variants. The specification of the implementation variants for Adapter and Observer design patterns are described in Section III. Section IV briefly describes how ePAD recovers design pattern instances. Section V reports and discusses the results of the performed exploratory study, while Section VI ends the paper presenting conclusion and future works.

II. DESIGN PATTERN RECOVERY APPROACHES SUPPORTING IMPLEMENTATION VARIANTS

Several approaches have been proposed to address the problem of design pattern recovery [1][3][8][9][11] [14][16]. In the following, we describe those declaring to support detection of different implementation variants. A complete discussion can be found in [3].

A bit-vector algorithm inspired by bio-informatics is adapted in [8] for identifying design pattern instances. Patterns and software systems to be analyzed are expressed in terms of string representations, which are formed by classes and relationships between them (association, aggregation, composition, instantiation, inheritance and dummy). Thus, the design pattern recovery problem is reduced to a problem of approximate string matching using bit-vectors. Two definition of design patterns are analyzed (i.e., Abstract Factory and Composite) and bit-vector algorithms on the string representations are provided to detect exact and approximate occurrences of the two design patterns in the program. In particular, approximate

occurrences of design patterns are obtained relaxing association relationship and entities participating in the instances and inheritance relationships. Moreover, relaxing the presence of all roles in the pattern instances this approach allows the detection of variants that miss something of the design pattern definitions. As an example, a micro-architecture having all the elements characterizing the definition of the Composite design pattern except a Leaf can be also considered an instance.

An approach based on the use of graphs has been proposed in [16] to automatically detect modified design patterns. The software and the design patterns to be retrieved are represented as graphs and matrices are used to represent important aspects of their static structure. Then, a graph similarity algorithm is employed to detect instances of candidate design patterns. The approach is able to detect modified pattern instances which are formed by attributes that follow the transitive property. This property is exploited by the similarity algorithm to detect modified pattern instances, e.g., variants of Adapter pattern admitting a class between classes playing the roles of Target and Adapter.

The aim of the approach proposed in [15] is to capture the intents of created patterns and provide a way to discover as many of their implementation variants as possible. The approach first establishes a simple meta-model of a program, consisting of a set of core elements and a set of relations among these elements. Then, first-order logic formulae are used to define design patterns and in particular their variants. Four GoF patterns are considered: Singleton, Factory Method, Abstract Factory, and Builder. These formulae allow to implement the detection method in various ways. In particular, for the Abstract Factory they consider that in practice a factory method often is not abstract, but returns a default instance, while the canonical definition imposes a strong constraint on code. The authors of this work also highlight that this variant is not allowed by other approaches implementing variations, like [16]. Moreover, another variant rarely considered for Abstract Factory involves a parameter passed to a factory method to parameterize the construction of a product. This is captured in [15], while other approaches, like PINOT [11] do not cover this variant.

The approach proposed in [6] exploits template matching method to recover design pattern instances from a software system by calculating their normalized cross correlation. A normalized cross correlation shows the degree of similarity between a design pattern and the part of a system. Similar to other approaches [8][15], it also encodes the pattern and system information into matrixes. Furthermore, it allows not only the exact matches of the instances from the source code of the system, but also the possibility of identifying the variants of pattern candidates. In particular, the variants are detected by admitting matrix matches with a high similarity score but less than 1. In other words, the approach allows to recover a candidate that matches the majority of the features of a design pattern definition because of the small angle between the candidate and the pattern template based on the normalized cross correlation [6]. The variants that can be detected are similar to the ones considered in [9][15].

A meta-modeling approach for the formalization of design patterns able to also recover variants has been proposed in [2]. In particular, the abstract syntax of UML (used to represent software models) is given in terms of a meta-notation extension of BNF, while structural and behavioral features of design patterns are specified by exploiting a first order predicate logic language. Thus, the problem of recovering instances of design patterns is reduced to the problem of verifying predicates, i.e., predicates are satisfied if and only if the input design conforms to the pattern. In order to admit alternatives and variants of design patterns, the approach allows to relax conditions by using keywords Optional, Alternatives and Depends on, when the structure of design patterns is specified. This approach is able to specify all the variants covered in [9][15].

The approach proposed in [12] exploits planar graphs to represent system and design patterns to be recovered in order to reduce the search space and consequently the recovery time. The key idea is that heuristics removing a few edges violating planarity do not significantly influence the recovery accuracy. The approach focused on the structural detection, thus not considering dynamic analysis. As for detection of design pattern variants, the approach allows to relax the presence of all roles in the pattern instances as done in [9]. As an example, for the Observer pattern they consider three instance types: (s, a, d, o), (s, o), and (s), where *s*, *a*, *d*, and *o* denote the class playing the role of Subject, the method attach, the method detach, and the class playing the role of Observer, respectively.

III. DESIGN PATTERN VARIANTS

The term *design pattern variant* usually refers to instances of a design pattern implemented in software systems that deviate from the standard specifications [18][19]).

In this section, we define 16 different implementation variants for the Adapter and Observer patterns, which represent the two patterns analyzed in the proposed exploratory study. They have been chosen since they can be considered as the representative of the structural and behavioral design pattern classes, respectively, and they have been used in the case studies discussed in the literature.

Starting from these definitions, we have considered ten implementation variants for the Adapter pattern. They differ from the canonical definition provided in [7] in several properties used to specify the design pattern, such as negative criteria, multi-level inheritance, type of the classes, references, and so on. In the following we introduce negative criteria and multi-level inheritance and then we list the definitions of the analyzed implementation variants.

The definition of a design pattern can be accompanied with *negative search criteria* (*NC*, for short) indicating the properties that are not allowed in context of a particular pattern [14]. As an example, for the object Adapter pattern defined by Gamma the negative criteria can be:

1. Adapter is not a sub class from the Adaptee class.
2. Adapter does not invoke methods of Target class

3. There do not exist inheritance or delegation relationships between Target and Adaptee.

For the class Adapter pattern we have the same *NC* except for the first one which is replaced by the following:

1'. Adaptee is not a sub class of Target class.

The presence of inheritance relationships in design pattern definitions introduces the possibility of having intermediate classes in the hierarchies with no role in the pattern. As an example, in the case of the object Adapter pattern there could be one or more classes between the Target and Adapter classes, which are involved in the pattern instance without having a role. We refer to *Multi-Level-Inheritance* (*MLI*) as the property of having this kind of classes in hierarchies.

A. Adapter design pattern variants

Two types of Adapter pattern have been defined in the literature: object Adapter pattern and class Adapter pattern [7]). Fig. 1(a) shows the structure of the object Adapter pattern in terms of a UML class diagram, which defines the participants to the pattern and how they collaborate. Fig. 1(b) shows the structure of the class Adapter pattern.

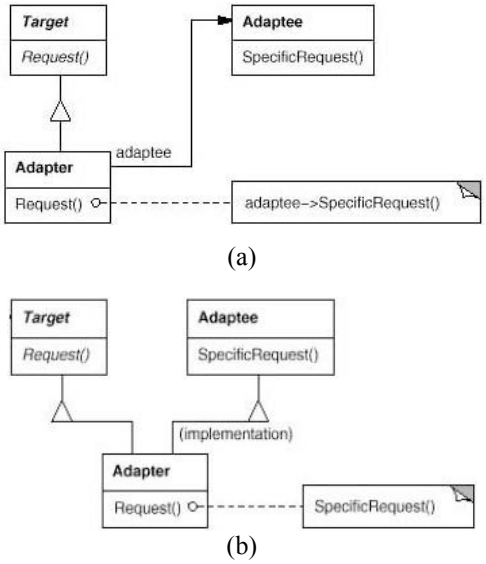

Figure 1. The structure of the object Adapter pattern (a) and class Adapter pattern (b).

In the following we list the different definitions of the Adapter pattern by reporting the differences with respect to the canonical definition provided in [7].

Definition A1. Strict definition of the object Adapter pattern as in the *structure* part of the Gamma definition shown in Fig. 1(a). In particular, no *MLI* between Target and Adapter classes, Target is an interface, Adapter and Adaptee are concrete classes.

Definition A2. Adds the three *NC* of object Adapter pattern described above to definition A1.

Definition A3. Extends definition A2 by admitting the presence of *MLI* between Target and Adapter classes.

Definition A4. Includes the same constraints specified for definition A3 except for the Target class which can be an abstract class also.

Definition A5. Extends definition A3 by relaxing the constraint on Adapter and Adaptee which can be abstract classes also.

Definition A6. Relaxes definition A3 by allowing the Target to be an abstract class, and the Adapter and Adaptee to be abstract classes (it merges the constraints included in definitions A4 and A5).

Definition A7. Extends definition A3 by adding a *Reference Constraint* (*RC*) between Adapter and Adaptee. Such a constraint specifies that the Adapter class contains an instance variable of type Adaptee.

Definition A8. Merges the constraints included in definitions A6 and A7.

Definition A9. Extends definition A8 by excluding *NC*.

Definition A10. Extends definition A6 by relaxing the delegation association between Adapter and Adaptee. Indeed, such a definition considers as valid both delegation and inheritance relationships between Adaptee and Adapter. In other words, it encompasses the class Adapter design pattern with the appropriate *NC*.

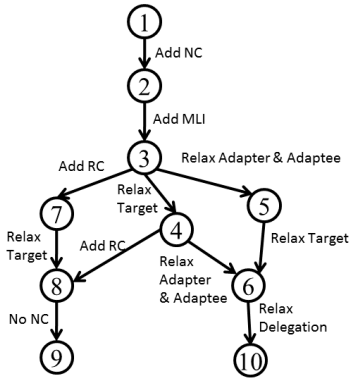

Figure 2. The graph showing the dependencies between the considered Adapter pattern definitions.

Fig. 2 depicts a graph describing the dependencies between the considered Adapter pattern definitions. In particular, a label *l* on the link connecting nodes *x* and *y* indicates that definition *y* is obtained from definition *x* by applying the modification *l*. As an example, the arrow between nodes 1 and 2 indicates that definition A2 is obtained from definition A1 through the addition of *NC*.

B. Observer design pattern variants

We have considered six definitions for Observer pattern. Fig. 3 shows the structure and the behavior of the Observer pattern in terms of a class diagram and a sequence diagram, respectively, while Fig. 4 depicts the graph describing the dependencies between the considered definitions.

Definition B1. Strict definition of the Observer pattern as in the structure and behavioral part of the Gamma *et al.* definition shown in Fig. 3 [7].

Definition B2. Relaxes definition B1 by adding *MLI*s between Subject and ConcreteSubject classes and between Observer and ConcreteObserver classes.

Definition B3. Extends definition B2 to make the presence of ConcreteSubject classes optional.

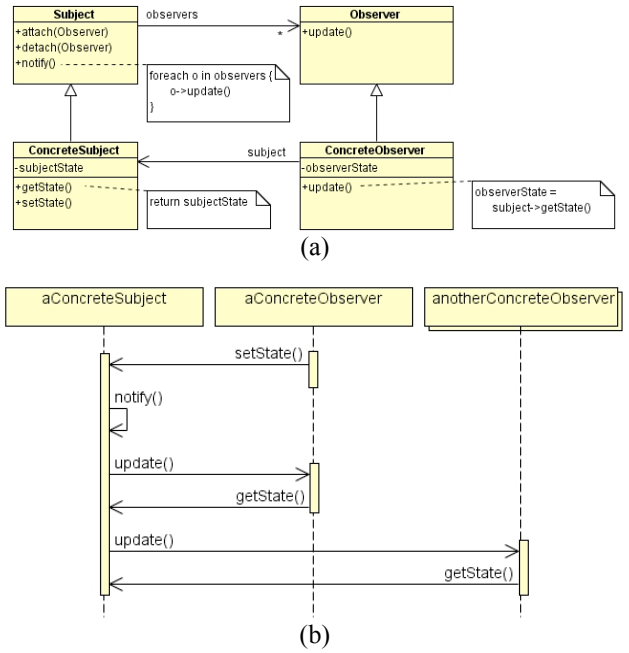

Figure 3. The structure (a) and the behavior (b) of the Observer pattern.

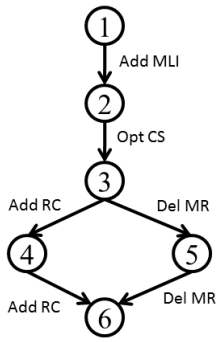

Figure 4. The graph showing the dependencies between the considered Observer pattern definitions.

Definition B4. Extends definition B3 by adding a RC between Observer and Subject.

Definition B5. Extends definition B3 by removing a Message Return (MR) check between ConcreteObserver and Subject/ConcreteSubject, which verifies whether the delegation getState() between the ConcreteObserver and the ConcreteSubject/Subject object is actual.

Definition B6. Extends definition B3 by adding a RC between Observer and Subject and removing a MR between ConcreteObserver and Subject/ConcreteSubject.

IV. THE EMPLOYED TOOL

This section briefly summarizes ePAD [5], the tool we employed in the case study to recover pattern instances.

ePAD identifies instances of structural and behavioral design patterns from Java code by applying static and dynamic analyses. Fig. 5 shows the activity diagram describing the recognition process of ePAD, where rectangles represent data and rounded rectangles represent

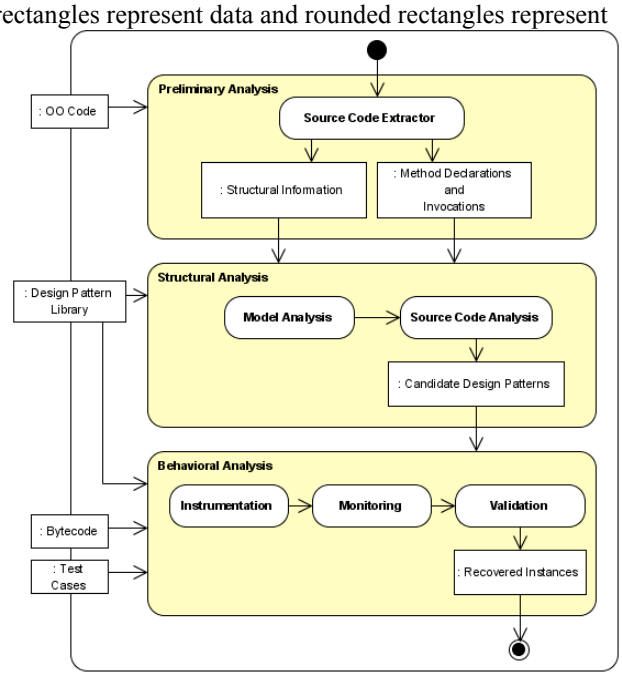

Figure 5. The recovery process of ePAD

process phases. During the *Preliminary Analysis* information proper to recover design pattern instances is extracted from input source code and stored in a repository by using the *Source Code Extractor*. In particular, class diagram information is exploited to construct the corresponding UML class diagram and stored to be used for the *Model Analysis*.

During the *Structural Analysis* the instances of design patterns are identified by analyzing the class diagram structure. This recovery process is organized in two steps. In the first step (*Model Analysis*), the candidate design patterns are identified at a coarse-grained level by analyzing the class diagram information obtained during the *Preliminary Analysis* and exploiting the *Design Pattern Library*. In the second step (*Source Code Analysis*), a fine-grained source code analyzer checks if the identified candidate patterns are correct instances or false positives. The definition of the recognition algorithms for these phases can be found in [3].

The set of candidate instances obtained from the *Structural Analysis* are given to the *Behavioral Analysis*, together with the specification of the pattern behaviors, and the executable program with a test suite (i.e., *Bytecode* and *Test Cases*). The goal is to identify the candidates having a behavior that complies with the pattern definition they are instance of. Details of the three steps can be found in [4].

V. CASE STUDIES

In the following we present the case studies we performed to investigate how design pattern implementation variants impact on the retrieval effectiveness of ePAD.

Table 1 and Table 2 summarizes the constraints and relaxations introduced for defining implementation variants of design patterns, their effect on the recovery process as well as the asymptotic complexity variation. Furthermore, the last column shows the expected impact on the performances of the proposed recovery process. The expected impact was than verified by applying ePAD on eight software systems: JHotDraw (JHD), versions 5.1 and 6.0b1; QuickUML 2001; JRefactory, version 2.6.24; JUnit, version 3.7; MapperXML, version 1.9.7; Lexi, version 0.1.1; and Nutch.

A. Results for Adapter pattern variants

Table 3 contains the recovery results for each Adapter pattern variant and each software system. In particular, we report on the number of recovered instance candidates, the recovery time, and the achieved precision (i.e., the fraction of recovered design pattern instances that are true). To evaluate precision, the occurrences retrieved by ePAD have been validated by two of the authors of this paper which were not involved in the implementation of ePAD. They reached a classification consensus on each retrieved design pattern instance by consulting the available documentation of the considered software and information from public repositories, such as P-MARt[1]. Note that for variant A10 we did not find any instance in all of the software systems we considered. This is because the class Adapter pattern is not widely used as the object Adapter pattern.

As for comparison of the results achieved with the different software systems, we can observe that the best precision values have been obtained with JHD. This can have a justification for the fact that JHD was originally developed to illustrate the good use of design patterns for designing and documenting systems. Furthermore, we can observe that:

- the most used implementation is variant A6 (i.e., the maximum number of actual instances recovered for the eight software systems);
- instances of variant A8 were characterized by the best precision value (in mean a precision of 0.46 for the eight software systems);
- instances of variants A1 and A10 were retrieved in less time (for all the software systems);
- instances of variant A9 allow to obtain the best trade-off between precision and time performances.

Regarding the impact of the employed constraints and relaxations in the implementation variant definitions, we report the achieved results in Table 4.

B. Results for Observer pattern variants

In Table 5 we report on the results achieved for each variant and software system for the Observer pattern. Furthermore, the precision values were obtained following the validation approach employed for the Adapter pattern.

TABLE 1. THE EXPECTED IMPACT OF THE DEFINED CONSTRAINTS AND RELAXATIONS FOR ADAPTER PATTERN VARIANTS ON PRECISION AND TIME PERFORMANCES

Constraint/ Relaxation	Effect on the recovery process	Asymptotic complexity variation	Expected impact on performances
Add NC A1 → A2	Three NC increase the complexity since further checks have to be verified	+ O(R), where R is the number of relations	+ Precision + Time
Add MLI A2 → A3	MLI between Target and Adapter classes increases the complexity since more classes have to be analyzed	+ O(I), where I is the number of inheritances	– Precision + Time
Relax Target A3 → A4 A5 → A6 A7 → A8	Relaxing the Target class type increases the complexity since it covers more cases	-	– Precision + Time
Relax Adapter & Adpatee A3 → A5 A4 → A6	Relaxing the Adapter and Adaptee class types increases the complexity since it covers more cases	-	– Precision + Time
Add RC A3 → A7 A4 → A8	The RC between Adapter and Adaptee classes increases the complexity since a further check has to be verified	+ O(V), where V is the number of instance variables	+ Precision + Time
No NC A8 → A9	The elimination of the three NC decreases the complexity	– O(R)	– Precision – Time
Relax Delegation A6 → A10	Relaxing the relatonship between Adaptee and Adapter classes increases the complexity since it covers more cases	+ O(R + I)	– Precision + Time

TABLE 2. THE EXPECTED IMPACT OF THE DEFINED CONSTRAINTS AND RELAXATIONS FOR OBSERVER PATTERN VARIANTS ON PRECISION AND TIME

Constraint/ Relaxation	Effect on the recovery process	Asymptotic complexity variation	Expected impact on performances
Add MLI B1 → B2	The MLI between Subject/Observer and ConcreteSubject/ ConcreteObserver classes increases the complexity since more classes have to be analyzed	+ O(I), where I is the number of inheritance	– Precision + Time
Opt CS B2 → B3	The Concrete Subjects verification increases the complexity since two cases have to be analyzed	-	– Precision + Time
Del MR B3 → B4 B5 → B6	The elimination of the MR check between the ConcreteObserver and the ConcreteSubject decreases the complexity	– O(R), where R is the number of relations	– Precision – Time
Add RC B3 → B5 B4 → B6	The RC between the Subject and the Observer classes increases the complexity since a further check has to be verified	+ O(V), where V is the number of instance variables	+ Precision +Time

The best precision values have been obtained with JHD 5.1, QuickUML, and Lexi. We can also highlight that:

- the most used implementation is the behavioral variant B4;
- instances of variant B4 was also characterized by the best precision value;

[1] http://www.ptidej.net/tools/designpatterns/

- instances of variant B1 was retrieved in less time;
- variant B4 allows to obtain the best trade-off between precision and time performances.

Regarding the impact of the employed constraints and relaxations in the implementation variant definitions, we report the achieved results in Table 6.

C. *Analysis of results*

Table 7 summarizes the results we achieved for each constraint/relaxation and design pattern considered in our study. It shows whether the expected impact of the constraints/relaxations given in Table 1 and Table 2 on the precision and time performances of ePAD is verified.

TABLE 3 THE RECOVERY RESULTS OBTAINED FOR THE ADAPTER PATTERN variants on eight open-source SOFTWARE SYSTEMS

		JUnit v3.7	**JHD v5.1**	**QuickUML 2001**	**MapperXML v1.9.7**	**JHD v6.0b1**	**JRefactory v2.6.24**	**Lexi v0.1.1**	**Nutch v0.4**	**TOTAL**
A1	Candidates	5	5	9	7	19	22	1	22	90
	Actual candid.	1	3	0	2	5	0	0	5	16
	Time (sec.)	2.4	2.9	4.6	8.5	18	69.6	3.3	21.3	130.6
	Precision	0.2	0.60	0	0.40	0.26	0	0	0.23	1.69
A2	Candidates	3	4	7	4	14	17	1	15	65
	Actual candid.	1	3	0	1	4	0	0	5	14
	Time (sec.)	2.6	4.40	5.4	9.5	25.2	91.2	3.4	24.2	165.9
	Precision	0.33	0.75	0	0.25	0.29	0	0	0.33	1.95
A3	Candidates	3	4	7	4	14	17	1	17	67
	Actual candid.	1	3	0	1	4	0	0	5	14
	Time (sec.)	2.6	3.9	5.2	9.4	43.2	88.7	3.4	24.1	180.5
	Precision	0.33	0.75	0	0.25	0.29	0	0	0.29	1.91
A4	Candidates	5	20	16	11	41	71	1	17	182
	Actual candid.	2	11	0	1	13	6	0	7	40
	Time (sec.)	2.4	3.5	6.2	10.6	37	101	3.3	25.8	189.8
	Precision	0.4	0.55	0	0.09	0.32	0.08	0	0.41	1.85
A5	Candidates	4	10	10	8	23	21	1	17	94
	Actual candid.	2	3	3	2	4	3	0	7	24
	Time (sec.)	2.4	4.2	4.8	9.2	29.5	85.8	3.3	24.9	164.1
	Precision	0.5	0.30	0.30	0.25	0.17	0.14	0	0.41	2.07
A6	Candidates	6	32	19	18	61	99	1	17	253
	Actual candid.	2	12	3	3	18	19	0	7	64
	Time (sec.)	2.6	5.7	6	11.1	43.5	104	3.4	25.8	202.1
	Precision	0.33	0.38	0.16	0.17	0.30	0.19	0	0.41	1.94
A7	Candidates	1	4	3	0	8	16	1	2	35
	Actual candid.	1	3	0	0	3	0	0	2	9
	Time (sec.)	2.6	5.4	5.1	8.2	32	89.2	3.4	25	170.9
	Precision	1	0.75	0	-	0.38	0	0	1	3.13
A8	Candidates	4	9	11	5	21	89	1	6	146
	Actual candid.	2	7	3	2	12	15	0	6	47
	Time (sec.)	3.5	8.6	7.1	11.9	56	131	3.9	25.2	247.2
	Precision	0.5	0.78	0.27	0.40	0.57	0.17	0	1	3.69
A9	Candidates	4	10	20	9	26	95	1	13	178
	Actual candid.	3	7	3	2	11	16	0	9	51
	Time (sec.)	3.4	7	6.7	9.3	38.6	116	3.8	28	212.8
	Precision	0.75	0.70	0.15	0.22	0.42	0.17	0	0.55	2.96
A10	Candidates	0	0	0	0	0	0	0	0	0
	Actual candid.	0	0	0	0	0	0	0	0	0
	Time (sec.)	2.3	3.5	4.6	8.9	22	62.1	3.3	16	122.7
	Precision	-	-	-	-	-	-	-	-	-

We can observe that the introduction of *NC* improves the precision by eliminating false positives but paying in terms of time performance. Thus, software engineers can add/remove these constraints to/from Adapter pattern definitions based on the trade-off between precision and time they are willing to accept.

With regard to *MLI*, we obtained conflicting results for the two considered design patterns. The analysis of our recovery approach revealed that two different visits of the class hierarchies are performed to recognize Adapter and Observer patterns. The hierarchy of the Adapter pattern is efficiently analyzed from subclasses to superclasses, i.e., from Adapter to Target, while the hierarchies of the Observer pattern are traversed from Subject to ConcreteSubject and from Observer to ConcreteObserver classes. This suggests that the impact of *MLI* relaxation on the recovery performances can be mitigated by the technique employed to analyze hierarchies. Our result can be generalized to other tools taking into account the approach used to represent and navigate class hierarchies.

The results about *Relax Target*, *Relax Adapter& Adaptee*, and *Relax Delegation* revealed that Adapter relaxation does not decrease the precision and does not increase time performances. This suggests that software engineers can use a less strict Adapter definition without worsening precision.

As expected the introduction of *Opt CS* relaxation deteriorates the performances. This suggests software engineers to carefully employ such a kind of relaxation for the Observer pattern.

The analysis performed for *Del MR* revealed that the implementation variants contained in the considered software systems do not employ the message return. Indeed, the elimination of these constraints did not influence the number of detected false positives. Furthermore, the decreasing of time performances is not verified since the elimination of *MR* constraints produces more candidates at the end of the first phase of the recovery process, and they are verified at a fine-grained level in the second phase. This suggests that software engineers should not remove *MR* constraints in the Observer design pattern definition.

With regard to the *Add RC* constraints, for the Adapter pattern we can provide similar considerations as for *NC*. Unexpectedly, this did not happen for Observer pattern. By deeply analyzing the achieved results, we observed that *Add RC* constraints are a subset of *MR* constraints. In particular, when the *MR* checks are not performed, the *Add RC* constraints are applied on a larger set of candidate instances and they eliminate truer positive instances. This suggests that software engineers should not include *RC* constraints in the Observer definitions without *MR* checks.

D. Validity evaluation

Several factors can bias the validity of this kind of empirical studies. The reliability of obtained results is highly influenced by the reliability of ePAD to recover design pattern instances. Other tools should be applied and compared with ePAD in the future to be more confident about the results. Unfortunately, the few available tools do not allow to change design pattern definitions. Moreover, the achieved results could also be influenced by the manual verification assessing the recovered instances. To mitigate this issue a comparison with the results achieved by other experimenters should be done.

TABLE 4. IMPACT OF CONSTRAINTS/RELAXATIONS FOR ADAPTER PATTERN

Constraint/ Relaxation	Impact on precision		Impact on time	
	Increased	Decreased	Increased	Decreased
Add NC A1 → A2	All the systems except MapperXML	MapperXML	All the systems except MapperXML	MapperXML
Add MLI A2 → A3	-	-	JHD 6.0	JHD 5.1
Relax Target A3 → A4	JUnit, JHD 6.0, JRefactory, Nutch	JHD 5.1, Mapper XML	QuickUML, MapperXML, JRefactory, Nutch	JUnit, JHD 5.1, JHD 6.0, Lexi
Relax Target A5 → A6	JHD 5.1, JHD 6.0, JRefactory	JUnit, QuickUML, MapperXML	All the systems	-
Relax Target A7 → A8	JHD 5.1, QuickUML, JHD 6.0, JRefactory	JUnit	All the systems	-
Relax Adapter & Adpatee A3 → A5	JUnit, QuickUML, JRefactory, Nutch	JHD 5.1, JHD 6.0	Nutch	All the systems except Nutch
Relax Adapter & Adpatee A4 → A6	QuickUML, MapperXMLJ Refactory	JUnit, JHD 5.1, JHD 6.0	JHD 5.1, JHD 6.0	All the systems except JHD 5.1 and JHD 6.0
Add RC A3 → A7	JUnit, Nutch JHD 6.0,	-	JHD 5.1	MapperXML, JHD 6.0
Add RC A4 → A8	All the systems	-	All the systems	-
No NC A8 → A9	JHD 5.1, QuickUML, MapperXML JHD 6.0, Nutch	JUnit	Nutch	All the systems except Nutch
Relax Delegation A6 → A10	*	*	-	All the systems

* we did not find any instance of A10

Other issues that could bias the results of our investigation regard the employed software systems and the considered kinds of design patterns.

TABLE 5 THE RECOVERY RESULTS OBTAINED FOR THE OBSERVER PATTERN VARIANTS ON EIGHT OPEN-SOURCE SOFTWARE SYSTEMS

		JUnit v3.7	JHD v5.1	QuickUML 2001	MapperXML v1.9.7	JHD v6.0b1	JRefactory v2.6.24	Lexi v0.1.1	Nutch v0.4	TOTAL
B1	Candidates	1	5	1	2	8	5	0	0	22
	Actual candid.	0	2	1	0	2	0	0	0	5
	Time (sec.)	2.4	5.3	4.8	9.9	29.5	100.1	3.1	3.7	158.8
	Precision	0	0.40	1	0	0.25	0	-	-	1.65
B2	Candidates	1	7	1	2	9	5	0	0	25
	Actual candid.	0	2	1	0	2	0	0	0	5
	Time (sec.)	2.5	12	5.8	12.9	112.9	138	3.1	3.7	290.9
	Precision	0	0.29	1	0	0.22	0	-	-	1.51
B3	Candidates	4	7	3	5	9	7	1	5	41
	Actual candid.	0	3	1	0	2	0	1	1	8
	Time (sec.)	3.7	22.9	10.9	18.3	158.6	259.5	3.3	7.2	484.4
	Precision	0	0.5	0.33	0	0.22	0	1	0.2	2.25
B4	Candidates	4	7	4	6	12	8	1	10	52
	Actual candid.	1	6	3	2	6	3	1	1	23
	Time (sec.)	5.6	35	16.5	21.7	189.8	256.4	7.4	7.4	539.8
	Precision	0.25	0.86	0.75	0.33	0.50	0.38	1	0.1	4.17
B5	Candidates	3	5	3	5	9	7	1	10	43
	Actual candid.	0	3	1	0	2	0	1	1	8
	Time (sec.)	3.4	28.4	10.7	23.3	253	341	3.2	11.4	674.4
	Precision	0	0.6	0.33	0	0.22	0	1	0.1	2.25
B6	Candidates	3	5	3	5	10	7	1	10	44
	Actual candid.	0	3	1	0	2	2	1	1	10
	Time (sec.)	3.5	26.3	12.7	24.1	238.3	336.4	3.3	10.6	655.2
	Precision	0	0.6	0.33	0	0.20	0.29	1	0.1	2.52

To mitigate this threat, we selected software systems with different sizes and widely employed in case studies assessing the accuracy of design pattern recovery approaches and tools. As for considered design patterns and their variants, since this is an exploratory study we decided to select one of the widely employed structural design patterns (i.e., Adapter) and one of the widely employed behavioral design patterns (i.e., Observer). Of course, in the future other software systems, design patterns, and pattern variants should be considered to allow the generalization of our results.

TABLE 6. IMPACT OF CONSTRAINTS/RELAXATIONS FOR OBSERVER PATTERN

Constraint/ Relaxation	Impact on precision		Impact on time	
	Increased	Decreased	Increased	Decreased
Add MLI B1 → B2		JHD 5.1 JHD 6.0	Remaining systems	Lexi, Nutch
Opt CS B2 → B3		JHD 5.1, QuickUML	All the systems	
Del MR B3 → B4	Remaining systems	Lexi, Nutch	Remaining systems	JRefactory
Del MR B5 → B6	JRefactory	-	-	-
Add RC B3 → B5	-	-	Remaining systems	JUnit, Lexi
Add RC B4 → B6	Lexi, Nutch	Remaining systems	MapperXML, JHD 6.0, JRefactory, Nutch	Remaining systems

TABLE 7 SUMMARY OF RESULTS

Constraint/ Relaxation		Adapter		Observer	
		Precision	Time	Precision	Time
Add NC		YES	YES	-	-
No NC		YES	YES	-	-
MLI		NO	NO	YES	YES
Relax	*Target*	NO	NO/YES/YES	-	-
	Adapter & Adaptee	NO	NO/YES	-	-
	Delegation	NO	NO	-	-
Add RC		YES	NO/YES	NO	NO
Opt CS		-	-	YES	YES
Del MR		-	-	NO	NO

VI. CONCLUSION AND FUTURE WORK

We have presented a preliminary study to show how design pattern implementation variants impact on the retrieval effectiveness of a design pattern recovery tool. In particular, we have highlighted the importance of analyzing design pattern variants to comprehend the mostly used implementations for Adapter and Observer patterns, the consequences of introducing a constraint or relaxation into the definition of Adapter and Observer patterns, and the impact of each constraint/relaxation on the recovery process.

As mentioned in Section V.D, in the future we intend to implement other design pattern variants and apply ePAD on other software systems. We intend to compare the results we achieved with those of other design pattern recovery tools able to implement the same design pattern variants. We also plan to extend the detection of implementation variants to those that exhibit a behavior different from the intended pattern behavior. They are particularly interesting to detect for refactoring purposes. Finally, we intend to verify how to exploit knowledge about design pattern variants during the execution of specific maintenance activities.

REFERENCES

[1] B. Bafandeh Mayvan and A. Rasoolzadegan, "Design pattern detection based on the graph theory", *Knowledge Based Systems*, 120, 2017, 211–225.

[2] I. Bayley and H. Zhu, "Formal Specification of the Variants and Behavioural Features of Design Patterns", *Journal of Systems and Software* 83(2), (2010) pp. 209-221.

[3] A. De Lucia, V. Deufemia, C. Gravino, M. Risi, "Detecting the Behavior of Design Patterns through Model Checking and Dynamic Analysis", *ACM Transactions on Software Engineering and Methodology*, 26(4), 2018, pp.1-41.

[4] A. De Lucia, V. Deufemia, C. Gravino, M. Risi, "Behavioral Pattern Identification through Visual Language Parsing and Code Instrumentation", in *Proc. of IEEE European Conference on Software Maintenance and Reengineering* (CSMR'09), 2009, pp. 99-108.

[5] A. De Lucia, V. Deufemia, C. Gravino, M. Risi, "An Eclipse plug-in for the Detection of Design Pattern Instances through Static and Dynamic Analysis", in *Proc. of IEEE International Conference on Software Maintenance* (ICSM'10), 2010, pp. 1-6.

[6] J. Dong, Y. Sun, and Y. Zhao, "Design Pattern Detection by Template Matching", in *Proc. of the 2008 ACM Symposium on Applied Computing* (*SAC'08)*, 2008, pp. 765-769.

[7] E. Gamma, R. Helm, R. Johnson, J. Vlissides, "*Design Patterns: Elements of Reusable Object-Oriented Software"*, Addison-Wesley, Menlo Park, CA, 1995.

[8] Y. Gueheneuc, S. Hamel, O. Kaczor, "Efficient identification of design patterns with bit-vector algorithm", in *Proc. of European Conf. on Software Maintenance and Reengineering* (*CSMR'06*), 2006, pp. 175-184.

[9] Y. Guéhéneuc, G. Antoniol, "DeMIMA: A Multilayered Approach for Design Pattern Identification", *IEEE Transactions on Software Engineering*, 34(5), 2008, pp. 667-684.

[10] C. Kramer, L. Prechelt, "Design Recovery by Automated Search for Structural Design Patterns in Object Oriented Software", in *Proc. of Working Conf. on Reverse Eng.* (*WCRE'96*), 1996, pp. 208-215.

[11] R. Olsson, N. Shi, "Reverse Engineering of Design Patterns from Java Source Code", in *Proc. of Intl Conference on Automated Software Engineering* (*ASE'06*), 2006, pp. 123-134.

[12] N. Pettersson and W. Löwe, "A Non-conservative Approach to Software Pattern Detection", in *Proc. of Intl Conference on Program Comprehension* (*ICPC'07*), 2007, pp. 189-198.

[13] N. Pettersson, W. Löwe, and J. Nivre, "Evaluation of Accuracy in Design Pattern Occurrence Detection", *IEEE Transactions on Software Engineering*, 36(4), 2010, pp. 575-590.

[14] I. Philippow, D. Streitferdt, M. Riebish, S. Naumann, "An Approach for Reverse Engineering of Design Patterns", *Software System Modeling*, 4(1), 2005, pp. 55-79.

[15] K. Stencel and P. Wegrzynowicz, "Detection of Diverse Design Pattern Variants", in *Proc. of Asia-Pacific Software Engineering Conference* (APSEC'08), 2008 pp.25-32.

[16] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, S. T. Halkidis, "Design Pattern Detection using Similarity Scoring", *IEEE Transactions on Software Engineering*, 32(11), 2006, pp. 896-909.

[17] D. Yu, Y. Zhang, and Z. Chen, " A comprehensive approach to the recovery of design pattern instances based on sub-patterns and method signatures", *J. Syst. Softw.*, 103, 2015, 1–16.

[18] A. Waheed, G. Rasool, S. Ubaid and F. Ghaffar, "Discovery of design patterns variants for quality software development", in *Proc. of Intl* Conf. on Intelligent Systems Engineering (ICISE), 2016, pp. 185-191.

[19] L. Wen-Jin, P. Ju-long, W. Kang-Jian. "Research on detecting design pattern variants from source code based on constraints." *Proc. of Intl* Journal of Hybrid Information Technology 8.5 (2015): 63-72.

[20] M. Zanoni, F. Arcelli Fontana, and F. Stella. "On applying machine learning techniques for design pattern detection", *J. Syst. Softw.*, 103, 2015, 102–117.