Automatic Recommendation of Software Design Patterns Using Anti-patterns in the Design Phase: A Case Study on Abstract Factory

Nadia Nahar* and Kazi Sakib†
Institute of Information Technology, University of Dhaka, Dhaka, Bangladesh
*bit0327@iit.du.ac.bd, †sakib@iit.du.ac.bd

***Abstract*—Anti-patterns, one of the reasons for software design problems, can be solved by applying proper design patterns. If anti-patterns are discovered in the design phase, this should lead an early pattern recommendation by using relationships between anti- and design patterns. This paper presents an idea called Anti-pattern based Design Pattern Recommender (ADPR), that uses design diagrams i.e. class and sequence diagrams to detect anti-patterns and recommend corresponding design patterns. First of all, anti-patterns relating to specific design patterns are analyzed. Those anti-patterns are detected in the faulty software design to identify the required design patterns. For assessment, a case study is shown along with the experimental result analysis. Initially, ADPR is prepared for recommendation of the Abstract Factory design pattern only, and compared to an existing code-based recommender. The comparative results are promising, as ADPR was successful for all cases of Abstract Factory.**

Keywords*—Software design, design pattern, anti-pattern, design pattern recommendation, abstract factory

I. Introduction

Design patterns formalize reusable solutions for common recurring problems, while anti-patterns are outcome of bad solutions degrading the quality of software. Design patterns are often mentioned as double-edged sword, selecting the right pattern can produce good-quality software while selecting a wrong one (anti-pattern) makes it disastrous [1]. Thus, which patterns to use in which situation, is a wise decision to take. On the contrary, mapping software usage scenario or user description with pattern intent is a manual and hectic task. However, this task can be made easier with assistance of pattern recommendation systems.

The recommendation of a proper design pattern is yet a faulty process due to the difficulties in connecting software information with design pattern intents. The software requirements do not contain possible design problems' indication, making it infeasible to identify the required patterns. However, anti-patterns can be detected after a faulty design is created from user requirements. Now, as every design pattern has its own context of design problems that it solves and every anti-pattern causes specific design problems, a relationship should exist between anti- and design patterns that can be beneficial in pattern recommendation.

This paper presents the idea of incorporating anti-pattern detection and design pattern recommendation in the software design phase. This idea is encapsulated in a tool named as Anti-pattern based Design Pattern Recommender (ADPR). The tool recommends appropriate patterns in two phases. The analysis of anti-patterns of particular design patterns is conducted in the first phase. For capturing the full anti-pattern information i.e. class structure, interactions, and linguistic relationships, the analysis is performed in three levels - structural, behavioral and semantic analysis. In the second phase, the inputted system is matched with those anti-patterns for recommending the related design patterns. This matching is also conducted in three levels similar as the levels of analysis - structural, behavioral and semantic matching. Based on the matched anti-patterns from these levels, the corresponding 'missing [2]' design patterns are recommended. ADPR is initially designed for the recommendation of Abstract Factory as it is one of the most popular patterns, and can be extended to the other patterns.

Research has been conducted for proposing pattern recommendation systems. However, those cannot provide a good precision due to the difficulty in logically defining the manual process of mapping human requirements with design pattern intents. The human requirements i.e. usage scenario, designers' answers to questions or cases residing in the knowledge base in Case Based Reasoning (CBR), have been inadequate to accurately extract the required design patterns because of the lack of focus on the design problems. Generally, these three approaches of design pattern recommendation can be found in the literature - textual matching of software usage scenario with design pattern intents [3], [4], [5], question answer session with designers [6], [7], and CBR [8], [9]. The first approach is inefficient to identify probable design problems of software as scenario does not contain design information. The generic questions of the second approach focuses more on design pattern features than design problems of particular software. In the third approach, cases of CBR does not store possible design problems of software. Oppositely, the field of anti-pattern detection identifies bad designs in software, assuring that successful detection of anti-patterns is possible [10], [11]. However, the usage of anti-pattern in the design phase for identifying correct design patterns is yet to be discovered.

A case study has been conducted for evaluating the applicability of the proposed approach. The case study is carried on a badly designed java project requiring Abstract Factory, named as $Painter$. Based on the step-by-step analysis on the project, Abstract Factory is recommended by the tool. This case study justifies the approach that, this recommendation process leads to the correct recommendations.

The validity of this approach is further justified by experimenting ADPR on the case of Abstract Factory design pattern. For this, the prototype of ADPR was implemented for Abstract Factory using java. Moreover, implementation of a prominent research on source based design pattern recommendation, proposed by Smith et al. [12], was also performed for the comparison. The dataset were created by gathering projects that require Abstract Factory, but intentionally has not been applied. The results are encouraging as ADPR provides better recommendation results in the design phase of software, compared to the source based one operating in the coding phase.

II. Related Work

In terms of recommending suitable patterns for software, the relationship establishment between the design pattern and anti-pattern is rare in the literature. Yet investigations have been conducted for proposing design pattern recommendation approaches from different perspectives as mentioned below. On the other hand, anti-pattern detection is a well-established research trend for successfully identifying anti-patterns to check whether the software design is bad.

A. Design Pattern Recommendation

As mentioned earlier, design pattern recommendation researches can be divided into three types – text-based search, question-answer session, and CBR. In text-based search, pattern intents are matched with the problem scenarios for identifying the design patterns that relate mostly to the software [3], [4], [5]. This intent matching is based on set of important words [3], text classification [4], or query text search using Information Retrieval (IR) techniques [5]. However, problem scenarios are ambiguous as written in human language; and are usually not written from a designer's point of view, making it impractical to identify possible design problems.

In question-answer based approach, designers are asked to answer some questions about the software and those answers lead to find the required patterns for that software [6], [7]. Here, the mapping from question-answers to design patterns is set by formulating Goal-Question-Metric (GQM) model [6], or ontology-based techniques [7]. The problem is that, the questions are often static or generic, and more related to design pattern features than software specific design problems.

In CBR, recommendations are given according to the previous experiences of pattern usage stored in a knowledge base in the form of cases [8], [9]. The retrieval of cases from the knowledge base is performed either using user provided class diagrams [8], or using inputted and reformulated problem descriptions [9]. Matching cases to identify required patterns are not feasible, as the cases do not focus on the design problems a software might have.

A few researches were conducted for recommending patterns which do not fall in any of the mentioned categories. Navarro et al. proposed a different recommendation system for suggesting additional patterns to the designer while a collection of patterns are already selected [13]. Thus, it may not be used for new software being developed. Kampffmeyer et al. presented a new ontology based formalization of the design patterns' intents making those focus on the problems rather than the solution structures [14]. However, the problem predicate and concept constraints, required by the recommendation tool, makes it's usage challenging. Both of these approaches require expertize of the designers to use those effectively.

The research question of this paper is to use anti-pattern knowledge for design pattern recommendation in the design-phase of software. The most related paper of this research is a code-level design pattern recommendation approach [12], where patterns are recommended dynamically during the code development phase. That research tried to relate anti-patterns with design patterns for recommendation. Anti-patterns were identified using structural and behavioral matching in the code, and required design patterns to mitigate those anti-patterns were recommended. However, design pattern recommendation in the coding phase is too late as the software has already been designed and needed to be changed after the recommendation.

B. Anti-pattern Detection

Anti-pattern detection is a rich area of research, that focuses on finding bad designs in software [15], [16], [17], [18]. Fourati et al. proposed an anti-pattern detection approach in design level using UML diagrams i.e. the class and sequence diagrams [10]. The detection was done based on some predefined threshold values of metrics, identified through structural, behavioral and semantic analysis. This prominent research assures that anti-pattern detection can be performed in the design phase. Another approach for anti-pattern detection was based on Support Vector Machines (SVM) [11], where the detection task was accomplished in three steps - metric specification, SVM classifier training and detection of anti-pattern occurrences. The concept of anti-pattern training has made any defined or newly defined anti-patterns detection possible, breaking the boundary of only detection of some well-established anti-patterns (e.g. Blob, Lava Flow, Poltergeists, etc.) [19].

As presented in subsection II-A, the existing approaches of design pattern recommendation in design phase use textual match with usage scenario, case match with knowledge base cases, or ask design pattern related generic questions to designers. These approaches cannot be the proper ways to recommend design patterns, as design patterns are used for mitigating design problems, and these do not focus on the system design problems. The single paper that focuses on design problems (anti-patterns), recommends design patterns in the coding phase, making its usage impractical.

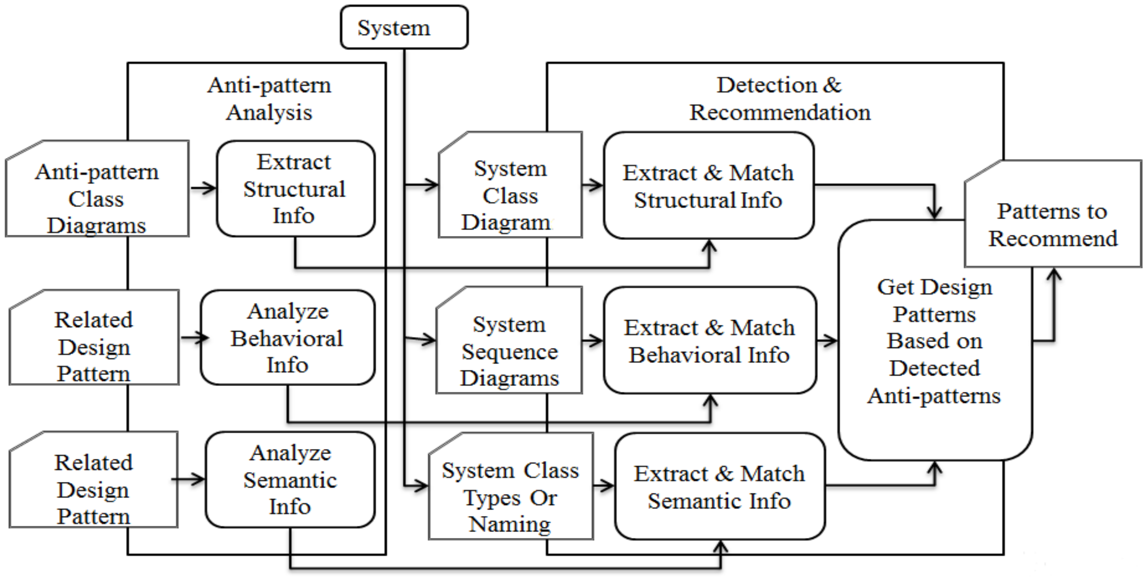
III. The Proposed Approach

The novelty of this research lies in identifying design problems of software for recommending appropriate design patterns, and in the design phase of software. Without having the analysis of bad designs (i.e. anti-patterns), suggesting correct design patterns is difficult. So, an idea is formalized, where the appropriate design patterns are suggested from identifying existing design problems, that reside as anti-patterns in the initial system design.

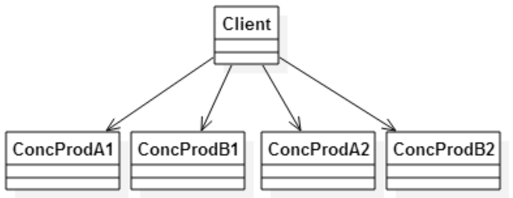
A. Overview of ADPR

Existence of an anti-pattern in a software design discloses that the design is not appropriate; the design can be improved by application of suitable design patterns. Thus, the detection of anti-patterns can lead to the recommendation of design patterns, if the anti-patterns could properly be mapped to their related design patterns.

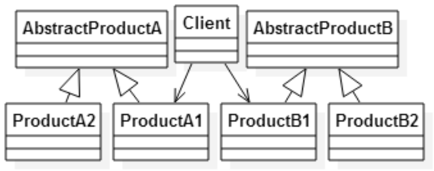
This idea is implemented as a system called Anti-pattern based Design Pattern Recommender (ADPR), which is initially designed for Abstract Factory design pattern. The top-level overview of ADPR is shown in Fig. 1. There are two


System
Anti-pattern Analysis
Anti-pattern Class Diagrams
Extract Structural Info
Related Design Pattern
Analyze Behavioral Info
Related Design Pattern
Analyze Semantic Info
Detection & Recommendation
System Class Diagram
Extract & Match Structural Info
System Sequence Diagrams
Extract & Match Behavioral Info
System Class Types Or Naming
Extract & Match Semantic Info
Get Design Patterns Based on Detected Anti-patterns
Patterns to Recommend
Fig. 1: Overview of ADPR

phases in the approach. At first the system analyzes the anti-patterns of particular design patterns. These anti-patterns do not necessarily be in the anti-patterns catalog like Blob, Lava Flow, etc[1]. These represent the '$missing$' design patterns [2] and their presence indicate that, a particular design pattern should have been used [20], [2], [12]. As shown in Fig. 1, in the second phase, the analyzed anti-patterns are detected in the initial system design and the corresponding design patterns to those matched anti-patterns are recommended. The detail of both these phases are described below.



(a) As Mentioned in [21]



(b) As Mentioned in [2]

Fig. 2: Anti-pattern Variants (Abstract Factory)

B. *Analysis of Anti-patterns*

To identify the $missing$ design patterns, the related anti-patterns are collected and analyzed first. The case of Abstract Factory is presented here as the usage example. Several anti-pattern variants of Abstract Factory may exist; initially, two of those are used (Fig. 2 [2], [21]) to show whether the proposed system works. In Fig. 2(a), there are two families of classes, $ConcreteProductA1$ ($ConcProdA1$), $ConcreteProductB1$ ($ConcProdB1$), and $ConcreteProductA2$ ($ConcProdA2$), $ConcreteProductB2$ ($ConcProdB2$). As determined by GoF, instead of being directly instantiated by the $Client$, these families should have been instantiated using abstract factories; this encourages the usage of Abstract Factory design pattern[2] in this case. Similarly in 2(b), $ProductA1$, $ProductB1$, and $ProductA2$, $ProductB2$ are two families of classes, which should not be directly instantiated by the $Client$. Thus, these two class designs represent the anti-patterns of Abstract Factory [2], [21].

These anti-patterns are analyzed and stored in the tool for further design level matching. Three levels of analysis are performed for ensuring the accurate capture of anti-pattern information - structural, behavioral and semantic (as shown in Fig. 1 'Anti-pattern Analysis' phase), similar to the design pattern analysis in [22].

The structural analysis concentrates on the structural characteristics of the anti-patterns. Similar structures of different anti-patterns can be found making this level of analysis inadequate. Thus, the behavioral analysis is provided for considering the behaviors of the anti-patterns along with the structure. One more level of validation is provided by the semantic analysis, as there can be cases where both structures and behaviors of different anti-patterns may match. Thus, these three levels of analysis ensure the proper refinement of the tool for detection of anti-patterns accurately.

Structural Analysis: The structure of an anti-pattern is defined by the relationships among the classes of it. Thus, class diagrams are used in this level [23] (as shown in Fig. 1, 'Anti-pattern Class Diagrams' are inputted to 'Extract Structural Info'), as those capture the different class-to-class relationships e.g. aggregation, generalization, association, etc. For keeping these relationship information, the structures are represented and stored in a form of $n \times n$ matrix of prime numbers as noted by Dong et al.[22] (for tracking cardinality

[1]"Anti Patterns Catalog," http://c2.com/cgi/wiki?AntiPatternsCatalog

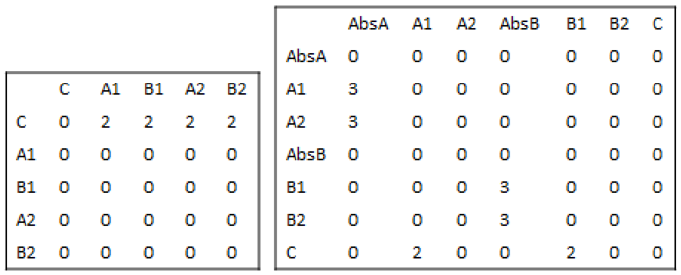
[2]Abstract Factory intent: "Provide an interface for creating families of related or dependent objects without specifying their concrete classes." [20]

of the relationships). Hence, this level takes the UML class information of anti-patterns as input and stores those in the form of matrices. For this, the class diagrams are converted to program readable format, XML and inputted to the tool.
In case of Abstract Factory, the class XMLs of the collected anti-pattern variants are provided to the analyzer, that creates and stores the structure matrices for each of the variants as shown in Fig. 3. The first matrix of Fig. 3 is generated from Fig. 2(a). Here,

- C, $A1$, $B1$, $A2$ and $B2$ represent $Client$, $ConcProdA1$, $ConcProdB1$, $ConcProdA2$ and $ConcProdB2$ respectively.
- The four association ($\xrightarrow{A}$) relations between $Client \xrightarrow{A} ConcProdA1$, $Client \xrightarrow{A} ConcProdB1$, $Client \xrightarrow{A} ConcProdA2$, $Client \xrightarrow{A} ConcProdB2$ in 2(a) are contained in the matrix using the prime number '2'[3].

Similarly, the second matrix of Fig. 3 is generated from 2(b), where,

- $AbsA$, $A1$, $A2$, $AbsB$, $B1$, $B2$, C represent $AbstractProductA$, $ProductA1$, $ProductA2$, $AbstractProductB$, $ProductB1$, $ProductB2$, $Client$ correspondingly.
- The four generalized ($\xrightarrow{G}$) relations ($ProductA1 \xrightarrow{G} AbstractProductA$, $ProductA2 \xrightarrow{G} AbstractProductA$, $ProductB1 \xrightarrow{G} AbstractProductB$, $ProductB2 \xrightarrow{G} AbstractProductB$) and two association relations ($Client \xrightarrow{A} ProductA1$, $Client \xrightarrow{A} ProductB1$) are stored in the matrix using prime number '3' and '2' consequently[3].



	C	A1	B1	A2	B2
C	0	2	2	2	2
A1	0	0	0	0	0
B1	0	0	0	0	0
A2	0	0	0	0	0
B2	0	0	0	0	0

	AbsA	A1	A2	AbsB	B1	B2	C
AbsA	0	0	0	0	0	0	0
A1	3	0	0	0	0	0	0
A2	3	0	0	0	0	0	0
AbsB	0	0	0	0	0	0	0
B1	0	0	0	3	0	0	0
B2	0	0	0	3	0	0	0
C	0	2	0	0	2	0	0

Fig. 3: Generated Matrices of Fig. 2

Behavioral Analysis: Behaviors of a system represent the dynamic characteristics (e.g. class execution sequence in run-time) of it. Now, it is logical to assume that the behaviors of a design pattern are inherited by it's anti-patterns, as the anti-patterns provide bad software structures compared to that pattern, but preserve the software behaviors. Thus, in behavioral analysis, the behaviors of the corresponding design patterns of anti-patterns are analyzed (Fig. 1, 'Related Design Pattern' leads to 'Analyze Behavioral Info').

[3]The determined prime number value of $Association$ is 2, $Generalization$ is 3, and $Aggregation$ is 5, similar as [12].

The behavioral feature of Abstract Factory is, there are families of classes, and these families are always used together [20]. Whenever such families of classes are found, that are always instantiated in the same execution path, and the classes of different families are instantiated in different execution paths, that system is required to use Abstract Factory [20].

Semantic Analysis: Semantic features of a system capture the logical relationships between classes (e.g. same types of classes in a system, classes that are always used together, etc.). Semantics basically relate the structural and behavioral aspects of the system (information of static structure with dynamic behavior). The semantic features of anti-patterns are also assumed to be the same as corresponding design patterns, as the logical relations among classes should not be changed, no matter how the system is being designed. Thus, similar as the behavioral analysis, related design patterns of anti-patterns are analyzed for capturing semantic information as shown in Fig. 1, 'Related Design Pattern' to 'Analyze Semantic Info'.
In Abstract Factory, classes of similar types form different families [20]. Therefore, the verification of behaviorally matched families are done by checking the types of the classes (identified from static structure) in families. Super-class information are used for this purpose, as classes having the same super-classes are generally of similar types; but there can be cases like Fig. 2 (a), where the design is bad enough to not even follow that OO convention. For those cases, similarity in the names of classes can give an indication of similar types.

C. Detection and Recommendation

Once the anti-patterns are analyzed based on corresponding design patterns, those could be detected in a faulty system design for recommending the patterns. Detection of anti-patterns needs three levels of matching similar to the analysis - structural, behavioral and semantic matchings (as shown in Fig. 1 'Detection & Recommendation' phase). If a system design is matched with an anti-pattern completely (structurally, behaviorally and semantically), only then the corresponding design pattern is recommended.

Structural Matching: The system structure is represented similarly as the matrix of anti-patterns using the system class diagram. The stored anti-patterns' structures (Fig. 3) are matched to the system's structure for finding whether any of those anti-patterns is present in the system (Fig. 1, from 'System Class Diagram' to 'Extract and Match Structural Info'). For this, the system matrix is matched with anti-patterns' matrices using naive approach, as the focus is on the accuracy rather the computational complexity or time. In this approach, matrices are matched using a brute force method where every permutation of the system matrix (permutation of nodes in the system graph) are taken and matched with the anti-pattern matrices. If no match is found, the detection is stopped and the other levels of matching are postponed. Otherwise, for at least one structural match, the behavioral matching is executed.

Behavioral Matching: Sequence diagrams are used in this level as those represent the dynamic interactions of classes in execution [23] (Fig. 1, 'System Sequence Diagrams' are inputted to 'Extract and Match Behavioral Info'). The $lifelines$

of a sequence diagram are the roles or object instances[4], and represent the classes in the same execution sequence. Thus, families of classes in Abstract Factory are identified from these $lifelines$, as classes of same families are supposed to be in the same execution sequence, and so in the same sequence diagram $lifelines$. For this, the UML sequence diagrams of the system are converted to XMLs first, and inputted to the tool. Then, the XMLs are parsed to identify the $lifelines$ and the corresponding classes of those are identified. Thus, the identified classes of each sequence diagram are marked to be in the same family.

Semantic Matching: Should a particular design pattern be recommended, is taken in the semantic matching step. In semantic matching for Abstract Factory, types of the classes are analyzed to validate the family information acquired from the behavioral matching as per the findings of semantic analysis (different classes of similar types form different families). A matrix containing the similar types of classes information is generated using the super-class relations. However, as mentioned earlier, sometimes the class-types could not be identified due to missing super-classes in a bad design (Fig. 2 (a)). For those cases, similarity in the names of the classes are analyzed to identify the same types (as shown in Fig. 1, 'System Class Types Or Naming' are used to 'Extract and Match Semantic Info'). The class names are split based on capital letters, and the parts are matched (For example, 'WoodenDoor' is split to 'Wooden', 'Door', and 'GlassDoor' is split to 'Glass', 'Door', and matched to each other). After the class types are determined, the mentioned type matrix is generated. Then, that matrix is used to analyze the classes in multiple families to test whether those are aligned to the assumption of Abstract Factory that, multiple families contain similar types of, but different classes.
Now, if the design is too bad to neither have super-classes nor similar names for the same types of classes, the approach will fail to generate type matrix and so, match semantics. Thus, for getting recommendation, the basic design principles should be followed by the designers. The semantic matching algorithm is shown in Algorithm 1.
For semantic matching, first of all the type matrix is generated (Algorithm 1 Line 8). As mentioned previously, it can be generated from super-class information (generalization relationship) or similar naming of classes. The type matrix is a 0,1 matrix, where the same type classes share value 1, and the others share value 0. Then, every sequences (class families) are compared to each others (Lines 9–13). The procedure COMPARESEQ is called for this reason. In COMPARESEQ, the duplicates in the sequences being compared are removed in Line 25. Then nested loops are executed for getting the positions of the classes of the sequences in the $type$ matrix using the class names list (cN) (Line 26–39). The value in those positions inside the $type$ matrix (0 or 1) is added to the seq matrix in Lines 41–42. After the calculation of the values in all the seq positions, $maxMatch$ between the sequences are identified in Lines 14–21. This $maxMatch$ is returned as the score of semantic matching. If the score value is >= 2, there is a valid semantic match.

[4]R. Perera, "The Basics & the Purpose of Sequence Diagrams - Part 1," http://creately.com/blog/diagrams/the-basics-the-purpose-of-sequence-diagrams-part-1/

Algorithm 1 Semantic Matching

```
1: $system$: System Matrix
2: $cN$: System Class Names
3: $behavioralMetric$: Behaviors of Anti-pattern (Sequence
   Diagram for Abstract Factory)
4: procedure MATCHSEMANTIC
5:     $seqs \leftarrow behavioralMetric.sequenceDiagrams$
6:     $size \leftarrow seqs.size()$
7:     $seq \leftarrow [size][size]$
8:     $type[cN.length][cN.length] \leftarrow$ GENTYPEMATRIX()
9:     for $i \leftarrow 0$ to $size$ do
10:        for $j \leftarrow i+1$ to $size$ do
11:            COMPARESEQ($seqs.get(i), seqs.get(j), i, j$)
12:        end for
13:    end for
14:    $maxMatch \leftarrow 0$
15:    for $i \leftarrow 0$ to $size$ do
16:        for $j \leftarrow 0$ to $size$ do
17:            if $maxMatch < seq[i][j]$ then
18:                $maxMatch \leftarrow seq[i][j]$
19:            end if
20:        end for
21:    end for
22:    return $maxMatch$
23: end procedure
24: procedure COMPARESEQ($s1, s2, p1, p2$)
25:    REMOVEDUPLICATES($s1, s2$)
26:    for $i \leftarrow 0$ to $s1.size()$ do
27:        for $j \leftarrow 0$ to $s2.size()$ do
28:            $s \leftarrow -1$, $d \leftarrow -1$
29:            for $k \leftarrow 0$ to $cN.length$ do
30:                if $s1.get(i) = cN.get(k)$ then
31:                    $s \leftarrow k$
32:                end if
33:                if $s2.get(j) = cN.get(k)$ then
34:                    $d \leftarrow k$
35:                end if
36:                if $s! = -1$ and $d! = -1$ then
37:                    $break$
38:                end if
39:            end for
40:            if $s! = -1$ and $d! = -1$ then
41:                $seq[p1][p2] \leftarrow seq[p1][p2] + type[s][d]$
42:                $seq[p2][p1] \leftarrow seq[p2][p1] + type[s][d]$
43:            end if
44:        end for
45:    end for
46: end procedure
```

IV. CASE STUDY ON "PAINTER", A PROJECT REQUIRING ABSTRACT FACTORY

For an initial assessment of the competency, ADPR was used on a sample java project named $Painter$ (Shown in Table I). This step-by-step study might increase the understanding of the tool as well as justify the feasibility of the approach.

It is assumed here that, the analysis of anti-patterns have already been performed. And thus, the tool has stored the required anti-patterns' information for the purpose of detecting those and recommending the corresponding design patterns for the inputted systems.

A. About *Painter*

The project, $Painter$ is a well-known example of Abstract Factory usage[5]. For testing the recommendation tool, the project is designed without implementing Abstract Factory (badly designed). The scenario of the project is as follows: "The $Paint$ can draw three types of $Shape$ - $Circle$, $Triangle$, or $Square$. The $Shape$s can be filled with three $Colors$ - Red, $Blue$, or $Green$. $Circle$s will be Red, $Triangle$s will be $Blue$, and $Square$s will be $Green$."

B. Structural Matching of *Painter*

As mentioned in 'Structural Matching' in subsection III-C, the system structure is to be matched with the anti-patterns' structure. For this, the initial class diagram of $Painter$, shown in Fig. 4, is inputted into the tool in XML format. This inputted XML is converted into a matrix of prime numbers for preserving the relationships between the classes (as instructed in [22]), as shown in Fig. 5. There are six association ($Paint \xrightarrow{A} Blue$, $Paint \xrightarrow{A} Green$, $Paint \xrightarrow{A} Red$, $Paint \xrightarrow{A} Square$, $Paint \xrightarrow{A} Triangle$, $Paint \xrightarrow{A} Circle$) and six generalization (($Blue \xrightarrow{G} IColor$, $Green \xrightarrow{G} IColor$, $Red \xrightarrow{G} IColor$, $Square \xrightarrow{G} IShape$, $Triangle \xrightarrow{G} IShape$, $Circle \xrightarrow{G} IShape$)) relationships in the diagram. These are fully preserved by putting value '2' in places of association and '3' in places of generalization[3].

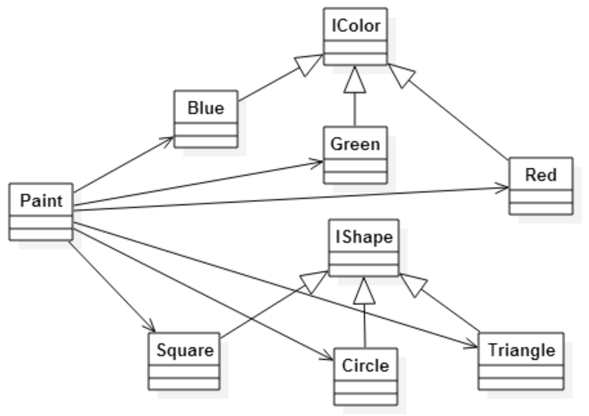
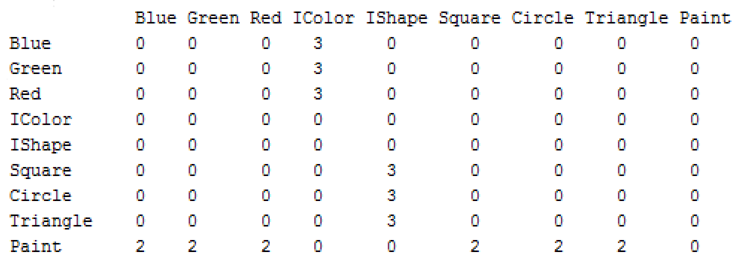


Fig. 4: Class Diagram of $Painter$

The anti-patterns' structures are assumed to be stored in the tool. Now, the structures of those stored anti-patterns are matched with the $Painter$ matrix using naive matrix matching. From Fig. 4 and Fig. 2 (a), a match is encountered. Thus, the structural matching is accomplished, and the tool will proceed to the next level of matching.

[5]"Design Pattern - Abstract Factory Pattern," http://www.tutorialspoint.com/design_pattern/abstract_factory_pattern.htm

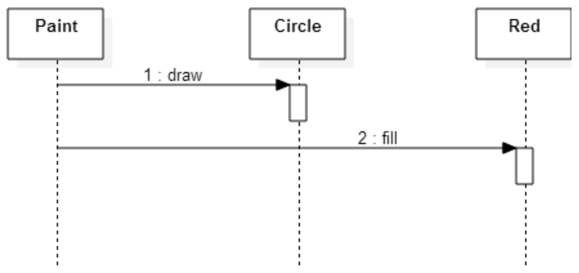


	Blue	Green	Red	IColor	IShape	Square	Circle	Triangle	Paint
Blue	0	0	0	3	0	0	0	0	0
Green	0	0	0	3	0	0	0	0	0
Red	0	0	0	3	0	0	0	0	0
IColor	0	0	0	0	0	0	0	0	0
IShape	0	0	0	0	0	0	0	0	0
Square	0	0	0	0	3	0	0	0	0
Circle	0	0	0	0	3	0	0	0	0
Triangle	0	0	0	0	3	0	0	0	0
Paint	2	2	2	0	0	2	2	2	0

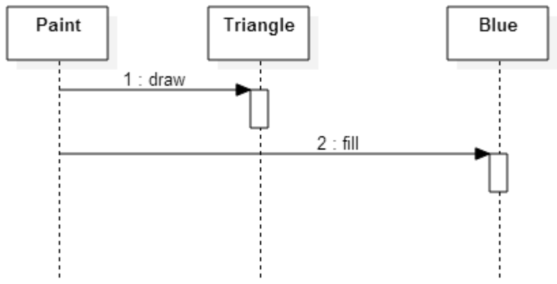
Fig. 5: Class Relation Matrix of $Painter$

C. Behavioral Matching of *Painter*

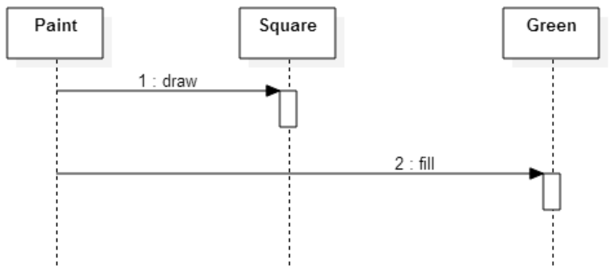
For behavioral matching, the information about the interactions between classes in execution is required. This information is extracted from the sequence diagrams. From the scenario of $Painter$, three sequence diagrams can be drawn (Fig. 6).



(a) Circle Is Red



(b) Triangle Is Blue



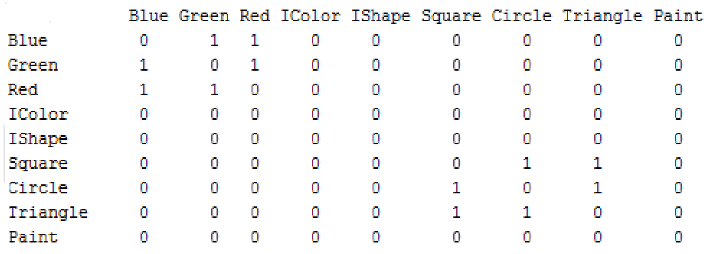
(c) Square Is Green

Fig. 6: Sequence Diagrams of $Painter$

The class families are identified from the $lifelines$ of these sequence diagrams. As, three sequence diagrams are inputted, three families are identified from those. The first family consists of $Paint$, $Circle$, and Red; the second family has the classes $Paint$, $Triangle$, and $Blue$; and the third family is comprised of $Paint$, $Square$, and $Green$.

D. Semantic Matching of *Painter*

The three families identified in the behavioral matching is validated in this level. First of all, the type matrix (as mentioned in subsection III-C 'Semantic Matching') is generated using the super-class information from the class relation matrix (Fig. 5). The type matrix is shown in Fig. 7. Situations can occur that the super-class information can be missing. For example, another variation of bad-designed class diagram can be created by the designer as shown in Fig. 8. It is noticeable here that, though the super-classes are missing, type matrix will still be generated from the similarity in the names of the same types of classes. Red$Color$, Blue$Color$, Green$Color$; and Circle$Shape$, Triangle$Shape$, Square$Shape$ are identified as same types. However, if the names of same types are not similar in this case, the approach will fail to generate the type matrix. For example - if the names of the classes are similar as Fig. 4, but the super-classes $IShape$ and $IColor$ are missing, then the approach will fail.



	Blue	Green	Red	IColor	IShape	Square	Circle	Triangle	Paint
Blue	0	1	1	0	0	0	0	0	0
Green	1	0	1	0	0	0	0	0	0
Red	1	1	0	0	0	0	0	0	0
IColor	0	0	0	0	0	0	0	0	0
IShape	0	0	0	0	0	0	0	0	0
Square	0	0	0	0	0	0	1	1	0
Circle	0	0	0	0	0	1	0	1	0
Triangle	0	0	0	0	0	1	1	0	0
Paint	0	0	0	0	0	0	0	0	0

Fig. 7: Type Matrix of $Painter$

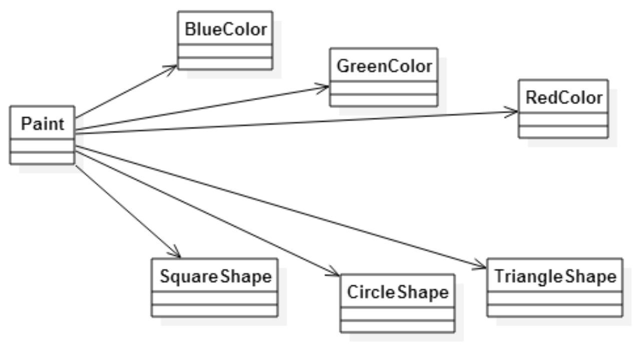


Fig. 8: Another Bad Class Diagram Example of $Painter$

After the type matrix is generated, the class families are analyzed to test whether different classes having the same types are situated in different families. Thus, the three identified families are analyzed here, and found that all three families contain classes of same types. $Circle$ (family-1), $Traiangle$ (family-2) and $Square$ (family-3) are of the same type, and similarly Red (family-1), $Blue$ (family-2) and $Green$ (family-3) are also same typed. So, the semantic matching ensures that the identified families from the behavioral matching are valid families.

All these three levels of matching indicate that the Abstract Factory design pattern is required to improve the project design. Thus, Abstract Factory is recommended for this project. This recommendation is obtained in the design phase of the project making it possible to re-design it, and provide a better design of the system.

V. Implementation and Result Analysis: for Abstract Factory

To assess the new approach, preliminary experiments have been conducted on Abstract Factory design pattern. A prototype of ADPR has been implemented in java for this purpose. The existing anti-pattern based pattern recommendation tool using source code [12] is also implemented for comparative analysis. For the justification of correct recommendations, GoF is followed [20].

A. Environmental Setup

As mentioned earlier, the ADPR prototype has been implemented in java. The equipments, used to develop the prototype are as follows:

- Eclipse Luna (4.4.1): java IDE for ADPR implementation
- StarUML Version-2.1.4: UML editor and XML converter

Four cases requiring Abstract Factory according to GoF, have been used as dataset. To test any occurrence of false positive, one project using Template pattern is used. The project source codes and UML diagrams are uploaded on GitHub [24]. The projects are shown in Table I.

TABLE I: Experimented Projects

Project Name	No. of Classes in Class Diagram	No. of Sequence Diagrams
CarDriver	8	2
GameScene	10	2
Painter	9	3
MazeGame	12	2
Trip	9	3

Before running ADPR on the sample project set, the XMLs are generated from the UMLs using StarUML to be used as input of the prototype. If the UMLs are not available, those can be produced from source code by reverse engineering in Visual Paradigm, a software design tool.

B. Comparative Analysis

For comparative analysis, the projects were run using both ADPR and the source based tool. The results of the experimentation are depicted in Table II, which shows that the code-based tool could detect two missing Abstract Factory patterns out of four. This is because, it assumed that the Abstract Factory has a behavioral aspect of having if-else or switch-case conditions for instantiating the families, which may not be always true (for example, class instantiations inside GUI onclick listener). On the other hand, ADPR was successful in all cases as the sequence diagrams do not assume the presence of any conditional operations, rather match the classes in one execution sequence. Both the tools did not produce any false-positive results.
The result identifies the fact that recommendations can be provided based on anti-patterns before the code development phase. Recommendation in the design phase gives opportunity

TABLE II: Results for Abstract Factory

Project Name	Recommend Abstract Factory		
	Code-Based	ADPR	GoF
CarDriver	Yes	Yes	Yes
GameScene	No	Yes	Yes
Painter	Yes	Yes	Yes
MazeGame	No	Yes	Yes
Trip	No	No	No

to correct the design of software which is not feasible in the coding phase. Thus, the results of ADPR are encouraging, as it could provide correct recommendations in the design phase, making the re-design of software possible.

VI. Conclusion

This paper introduces a new idea to recommend design patterns using anti-patterns. A tool is proposed named ADPR, where anti-pattern detection is utilized for recommendation of appropriate design patterns in the software design phase.
The recommendation task is executed in two phases; analysis of anti-patterns is performed in the first phase, and in the next phase, anti-patterns are detected and design patterns are recommended. For anti-pattern analysis in the first phase, anti-patterns of particular design patterns are collected and analyzed in three levels - structural, behavioral, and semantic. Then in the second phase, the identified anti-patterns are matched with system designs for recommending corresponding design patterns using the similar three levels of matching.
A case study on a sample java project evaluates the applicability of the approach. The tool was initially implemented for Abstract Factory only. A comparative analysis with an existing code based tool showed that, ADPR could correctly recommend design patterns in the design phase rather in the coding phase.
As currently the tool is developed for Abstract Factory, the future direction lies in extending it to the other design patterns incrementally, and generalizing the process.

References

[1] N. Bautista, "A Beginners Guide to Design Patterns," http://code.tutsplus.com/articles/a-beginners-guide-to-design-patterns--net-12752, accessed: 2015-01-01.

[2] C. Jebelean, "Automatic Detection of Missing Abstract-Factory Design Pattern in Object-Oriented Code," in *Proceedings of the International Conference on Technical Informatics*, 2004.

[3] Y.-G. Guéhéneuc and R. Mustapha, "A Simple Recommender System for Design Patterns," in *Proceedings of the 1st EuroPLoP Focus Group on Pattern Repositories*, 2007.

[4] S. M. H. Hasheminejad and S. Jalili, "Design Patterns Selection: An Automatic Two-phase Method," *Journal of Systems and Software, Elsevier*, vol. 85, no. 2, pp. 408–424, 2012.

[5] S. Suresh, M. Naidu, S. A. Kiran, and P. Tathawade, "Design Pattern Recommendation System: a Methodology, Data Model and Algorithms," in *Proceedings of the International Conference on Computational Techniques and Artificial Intelligence (ICCTAI)*, 2011.

[6] F. Palma, H. Farzin, Y.-G. Guéhéneuc, and N. Moha, "Recommendation System for Design Patterns in Software Development: An DPR Overview," in *Proceedings of the 3rd International Workshop on Recommendation Systems for Software Engineering*. IEEE, 2012, pp. 1–5.

[7] L. Pavlič, V. Podgorelec, and M. Heričko, "A Question-based Design Pattern Advisement Approach," *Computer Science and Information Systems*, vol. 11, no. 2, pp. 645–664, 2014.

[8] P. Gomes, F. C. Pereira, P. Paiva, N. Seco, P. Carreiro, J. L. Ferreira, and C. Bento, "Using CBR for Automation of Software Design Patterns," *Advances in Case-Based Reasoning, Springer Berlin Heidelberg*, vol. 2416, pp. 534–548, 2002.

[9] W. Muangon and S. Intakosum, "Case-based Reasoning for Design Patterns Searching System," *International Journal of Computer Applications*, vol. 70, no. 26, pp. 16–24, 2013.

[10] R. Fourati, N. Bouassida, and H. B. Abdallah, "A Metric-Based Approach for Anti-pattern Detection in UML Designs," *Studies in Computational Intelligence, Springer Berlin Heidelberg*, vol. 364, pp. 17–33, 2011.

[11] A. Maiga, N. Ali, N. Bhattacharya, A. Sabané, Y.-G. Guéhéneuc, G. Antoniol, and E. Aïmeur, "Support Vector Machines for Anti-pattern Detection," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2012, pp. 278–281.

[12] S. Smith and D. R. Plante, "Dynamically Recommending Design Patterns," in *Proceedings of the 24th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, 2012, pp. 499–504.

[13] I. Navarro, P. Díaz, and A. Malizia, "A Recommendation System to Support Design Patterns Selection," in *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2010, pp. 269–270.

[14] H. Kampffmeyer and S. Zschaler, "Finding the Pattern You Need: The Design Pattern Intent Ontology," *Model Driven Engineering Languages and Systems, Springer Berlin Heidelberg*, vol. 4735, pp. 211–225, 2007.

[15] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur, "Decor: A Method for the Specification and Detection of Code and Design Smells," *IEEE Transactions on Software Engineering, IEEE*, vol. 36, no. 1, pp. 20–36, 2010.

[16] T. Feng, J. Zhang, H. Wang, and X. Wang, "Software Design Improvement through Anti-patterns Identification," in *Proceedings of the 20th IEEE International Conference on Software Maintenance*. IEEE, 2004, p. 524.

[17] A. Maiga, N. Ali, N. Bhattacharya, A. Sabane, Y.-G. Guéhéneuc, and E. Aimeur, "SMURF: A SVM-based Incremental Anti-pattern Detection Approach," in *Proceedings of the 19th Working Conference on Reverse Engineering (WCRE)*. IEEE, 2012, pp. 466–475.

[18] V. Cortellessa, A. Di Marco, R. Eramo, A. Pierantonio, and C. Trubiani, "Digging into UML Models to Remove Performance Antipatterns," in *Proceedings of the 32nd ICSE Workshop on Quantitative Stochastic Models in the Verification and Design of Software Systems*. ACM, 2010, pp. 9–16.

[19] W. J. Brown, H. W. McCormick, T. J. Mowbray, and R. C. Malveau, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley New York, 1998.

[20] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, 1994.

[21] A. Jarvi, "Abstract Factory: 2005," http://staff.cs.utu.fi/kurssit/Programming-III/AbstractFactory(10).pdf, accessed: 2015-01-03.

[22] J. Dong, D. S. Lad, and Y. Zhao, "DP-Miner: Design Pattern Discovery Using Matrix," in *Proceedings of the 14th Annual IEEE International Conference and Workshops on Engineering of Computer-Based Systems (ECBS)*. IEEE, 2007, pp. 371–380.

[23] H. Zhu and I. Bayley, "An Algebra of Design Patterns," *ACM Transactions on Software Engineering and Methodology (TOSEM), ACM*, vol. 22, no. 3, p. 23, 2013.

[24] N. Nahar, "NadiaIT/ADPR-dataset: 2015," https://github.com/NadiaIT/ADPR-dataset, accessed: 2015-06-05.