ACDPR: A Recommendation System for the Creational Design Patterns Using Anti-patterns

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*—Recommendation of creational design patterns is not an easy task due to the similarities in their intents. Relationships with anti-patterns can play a significant role here. However, the logical definition and characterization of these anti-patterns, categorized by the design patterns, is still missing. The contributions of this paper are to identify and characterize the anti-patterns that reside as alternative solutions to the creational design patterns, and detect those anti-patterns in software designs to recommend the correct design patterns. After the anti-patterns are defined through the analysis of their structure, behavior and semantic, a tool named Anti-pattern based Creational Design Pattern Recommender (ACDPR) is developed. ACDPR uses the findings of the analysis to detect the anti-patterns in software design, performing structural, behavioral and semantic matchings. The outcome of these matching levels are used to calculate a score for each pattern, and based on these scores, creational design patterns are recommended. The justification of the approach is done by running the tool in 21 software, that results in a precision of 1, recall of 0.95, and F-measure of 0.97.**

I. Introduction

Design patterns are the proven solutions to common recurring problems, while anti-patterns are bad solutions. *Creational patterns* is one of the three categories (*creational, structural, behavioral*) of design patterns. These are quite inter-related and possess similar intents, making it difficult to select the appropriate one, e.g., confusion in selecting Factory Method or Abstract Factory. This selection task is made easier with assistance of pattern recommenders [1].

Like any design pattern, recommendation of creational patterns is not an easy task due to the difficulties in logically defining the manual process of pattern selection. As software requirements do not express the possible design problems of software, it is not feasible to identify the required design patterns to solve the problems. Now, as the existence of certain anti-patterns in design indicates that it can be improved by applying particular design patterns, anti-pattern detection can be useful in design pattern recommendation. Even so, relationship of anti-patterns with design patterns and characterization of those are yet to be established.

Different approaches of design pattern recommendation can be found in the literature - textual matching of software usage scenario with design pattern intents [1], [2], question answer session [3], [4], Case Based Reasoning (CBR) [5], and anti-pattern detection [6], [7]. The textual matching approach is not appropriate for the creational patterns due to the intent similarities between the patterns. This approach is questionable for the other patterns also, as the scenarios do not contain design information leading to pattern recommendations without even knowing the design problems of software. In the second approach, generic design pattern related questions are asked to the designers that lack focus on the individual pattern properties as well as the design problems. The third approach is also a generic one, and misses the possible design problems of software in CBR stored cases. Thus, matching a new software description with the stored cases does not necessarily assure the applicability of the patterns on that new software. Oppositely, the category of anti-pattern detection suggests design patterns against the detected anti-patterns in software. However, the robustness of the works is questionable, as the anti-pattern analysis was done only for Abstract Factory (in [7]), and Singleton (in [6]). Also, [6] recommends patterns in coding phase, which is too late as the software has already been developed.

This paper proposes an approach to recommend creational design patterns using anti-pattern information in the software design phase. As anti-patterns are the bad alternative solutions to the design patterns, those are termed as $missing$ design patterns [7]. The contributions of this paper are to logically derive the characteristics of the $missing$ creational design patterns, and to recommend the creational patterns based on the detection of these derived characteristics in a software design. Three levels of analysis are performed - structural, behavioral and semantic, for deriving the anti-pattern information. The outcome of these analysis is encapsulated in a tool named as Anti-pattern based Creational Design Pattern Recommender (ACDPR). For generating the recommendations, it conducts three levels of matchings (structural, behavioral and semantic) to detect the analyzed anti-pattern characteristics in the software design. As the detection of anti-patterns of a particular pattern represents the $missing$ design pattern, each design patterns are given a score, calculated using the results of the matchings. These scores determine which patterns are to be recommended.

Validity of the approach is justified by experimenting ACDPR on software designs requiring creational patterns. For this, ACDPR was implemented in java, and 21 software were used as dataset. For this dataset, ACDPR provides a precision of 1, recall of 0.95, and F-measure of 0.97.

II. Recommendation Approach

An overview of the recommendation approach is shown in Fig. 1. First, anti-patterns corresponding to creational pattens

are identified and analyzed. Based on this analysis, a framework called Anti-pattern based Creational Design Pattern Recommender (ACDPR) is devised. Steps of ACDPR are – matching anti-patterns' structure, behavior, and semantic using design diagrams; calculating score of each design pattern based on detection of corresponding anti-patterns; and recommending patterns based on the obtained scores.

=

Figure 1: Overview of the Recommendation Approach

A. Analysis and Matching of Anti-patterns

The anti-patterns of the five creational patterns are derived and analyzed to identify the structural, behavioral, and semantic properties. Also, the mechanism of matching these properties with design diagrams are described.

1) $missing$ Abstract Factory: The anti-pattern of Abstract Factory arises when groups of classes are instantiated directly without using factories for the instantiation [7].

a) Structure: Several structural variations of Abstract Factory anti-pattern exist in form of class diagrams[1].

For the structural matching, these variants are stored in the forms of $n \times n$ matrices of prime numbers (as in [7]). The software structure is represented as similar matrix using software class diagram. Finally, stored anti-pattern matrices are matched to the software matrix (similar to [6]).

b) Behavior: In Abstract Factory, there are families of classes, and these families are always used together. While, the classes of a family are instantiated in one execution path, classes of different families are executed in different paths (conditional instantiation of group of classes).

As classes of same families are supposed to be in the same execution sequence, families of classes are identified from the $lifelines$ of sequence diagrams. Here, existence of multiple sequence diagrams simply assure that there are multiple execution paths, and the classes of each execution path are considered to be in one family.

c) Semantic: Here, classes of similar types form different families. To be more specific, the multiple families are comprised with different classes of same types.

For semantic matching, a matrix containing the similar types of classes information is generated using the super-class relations. In absence of super-classes, similarity in the names of the classes are analyzed to identify the same types (similar as [7]). Finally, the matrix is used to analyze the classes in multiple families to test whether those are aligned to the requirements of Abstract Factory.

2) $missing$ Factory Method: The anti-pattern of Factory Method is similar to the anti-pattern of Abstract Factory, but with absence of class families.

a) Structure: The structural variations of Factory Method are uploaded on GitHub[1]. The structural matching process of this pattern is same as Abstract Factory.

[1]https://github.com/NadiaIT/ACDPRAntiPatterns

b) Behavior: In Factory Method, different classes are instantiated in different execution paths (conditional instantiation). Unlike Abstract Factory, where families of classes are instantiated in different execution paths, here only single classes are instantiated in those execution paths.

The behavioral matching is also similar to Abstract Factory. The identified classes from $lifelines$ of each sequence diagram are marked to be in the same execution paths.

c) Semantic: Classes instantiated in different execution paths are of the same type in Factory Method. It is important to note that, when such single classes are found, Factory Method is applicable; but if there are multiple classes (family of classes), Abstract Factory has the potential to be used.

The types of classes are analyzed to validate the single class group. This step is also similar as Abstract Factory. The only difference is, for Abstract Factory, at least two types of classes exist in families, and for Factory Method, only one type classes are in different execution sequences.

3) $missing$ Builder: Builder was created to find a solution to the problem of Telescoping Constructor anti-pattern. Thus, the existence of Telescoping Constructor represents the $missing$ Builder pattern. The structure and behavior are enough for the perception of Telescoping Constructor; thus, the semantic properties of $missing$ Builder is not needed.

a) Structure: Anti-pattern of Builder is concerned with single class rather than multiple ones. So, despite covering the whole class relationships structure, it requires to analyze the internal structure of classes individually. The structure is, there are multiple constructors of a class with different parameters. Existence of these multiple different parameterized constructors assure that the single object has different representations revealing the potential of using Builder.

Every class is inspected one-by-one to identify existence of multiple constructors. Classes having at least two constructors are the potential classes for applying Builder.

b) Behavior: The behavior lies in the parameter lists of the multi-constructor classes, where with inclusion of every constructor, a new parameter is introduced. This behavior shows that the object can be divided into individual parts based on those parameters, as required by Builder.

Thus, classes having multiple constructors are examined to identify if this parameter list pattern can be found.

4) $missing$ Prototype: The $missing$ Prototype basically reveals the usage of multiple instances of same objects, which recommends the cloning of those objects. Similar to Builder, semantic of $missing$ Prototype is not required.

a) Structure: The $prototype$ classes contain clone methods to copy instances of own-selves, that can be invoked by other classes. A clone method is called by another class only when it requires multiple instances of the $prototype$ class. Thus, the structure focuses on these other classes to find whether multiple variables of a class type are present.

The structural matching is done by checking if a class contains multiple variables having another class type.

b) Behavior: As structure defines, in Prototype, a class must have multiple variables having the type of $prototype$ class. The behavior verifies that, truly $prototype$ classes are instantiated in those variables (in case of inheritance, sub-classes can get instantiated in a variable of super-class type).

The behavior of the $missing$ Prototype are two-fold. In the first case, the identified class variable type (the class having multiple occurrences) is not of a super-class. In this case the variables will not contain instantiations of any other classes (sub-classes); but of that class type only. Thus, the multiple instantiations are of a single class, that is the $prototype$ class. In the second case, if the target class is a super class, the variables may contain instantiations of sub-classes. Thus, it is possible that different classes are instantiated in a generalized variable type. For the verification of the instantiated classes, sequence diagrams are used. The diagrams having the $instantiator$ class as a lifeline are analyzed to check, whether multiple children of the target class are connected to the $instantiator$ class. If multiple children are not connected, the instantiations are of only one class (the single connected child), making it the $prototype$ class. Otherwise, the multiple instantiations of one class cannot be confirmed.

5) $missing$ Singleton: In proper application of Singleton pattern, $singleton$ class has the responsibility of keeping one single instance of it, ensuring central control to object creation. In the anti-pattern, a conditional checking is performed before instantiating the $singleton$ class every time by the classes that require the $singleton$ object [6]. There is no structural property of $missing$ Singleton.

a) Behavior: The behavior is to contain conditional checking before instantiating a module to verify whether it has already been instantiated or not.

This behavior is detected using activity diagrams. First, WordNet is used to find similar terms as 'instantiate', 'create', 'initiate', 'load', etc. in any action node of the diagrams. If found, its connectivities are checked to identify whether any decision node is connected to that action node. The connectivities of the decision nodes are also examined to find the full structure of conditional checking.

b) Semantic: The semantic matching verifies whether the identified module name is present in the list of classes.

B. Score Calculation and Design Pattern Recommendation

Based on the match or mismatch of the structure, behavior, and semantic, each design pattern obtains a score between 0 to 1. The formula of the score calculation is –

$$score_i = \frac{\alpha \cdot str_i + \beta \cdot beh_i + \gamma \cdot sem_i}{\alpha + \beta + \gamma} \quad (1)$$

where,

- str, beh, sem are outcome of structural, behavioral and semantic matching (0 for mismatch, 1 for match);
- α, β, γ are the weights of the structural, behavioral and semantic properties of the anti-patterns respectively;
- and, $i \in \{1, ..., n\}$, for n design patterns

Now, for some $missing$ patterns, all these three properties are not present. In those cases, the weight of that property is set to 0 ($\gamma = 0$ for Builder, Prototype; $\alpha = 0$ for Singleton). For the other cases, the weight is set based on the significance of that property. Structural match provides a general level of matching, behavioral match refines the anti-pattern's presence by specifying it to some extent, and finally semantic match concludes the existence of the anti-pattern by considering the finest details. Thus, the relation between the weights of these levels are - $\alpha < \beta < \gamma$. For the sake of simplicity, the weight of a level is given a value, twice the value of the previous level's weight; $\alpha = 1, \beta = 2, \gamma = 4$.

If a design pattern gets the highest score of 1, it perfectly fits in that software design, and thus is recommended. For the other cases, mismatch occurred in some level of matching. This mismatch can be a reason of failure to extract the software design properly to match that anti-pattern, incomplete design diagrams, or the anti-pattern is truly not present in the design. Thus, if two of the properties match, and one mismatch, their is a chance that the anti-pattern might exist in the software. Thus, for a partial matching (score < 1), the design patterns are suggested based on a threshold value. As already stated, design patterns can be suggested if multiple levels match; making an appropriate threshold value 0.43 ($(\alpha + \beta) \div (\alpha + \beta + \gamma) = 0.43$). However, if only the semantic is matched, it will still have a value greater than the threshold ($\gamma \div (\alpha + \beta + \gamma) = 0.57 > 0.43$). However, it can never happen, as the semantic matching refines the behavioral matching; if the behavior matching is failed, the semantic matching cannot get a positive response.

This threshold value is only appropriate for patterns having all three levels of matching, but Builder, Prototype and Singleton only have two levels of matching. Thus, the suggestions of these patterns should be given if one of the levels is matched successfully, making an appropriate threshold value of 0.3 ($\beta \div (\beta + \gamma) \approx 0.3$ for Builder, Prototype; and $\alpha \div (\alpha + \beta) \approx 0.3$ for Singleton). This value is also applicable for the patterns having the three levels of matching ($\alpha \div (\alpha + \beta + \gamma) = 0.14 < 0.3$, $\beta \div (\alpha + \beta + \gamma) = 0.29 < 0.3$). Thus, for a score greater than the threshold value, 0.3, a design pattern is suggested.

III. Implementation and Result Analysis

ACDPR has been implemented in java. 21 projects requiring different creational patterns have been used as dataset. The project design diagrams are uploaded on GitHub (https://github.com/NadiaIT/ACDPR-dataset). The expected recommendation of the projects according to GoF are - Abstract Factory (project 1-4), Factory Method (5-8), Builder (9-13), Prototype (14-17), and Singleton (18-21).

For the analysis of the results, the dataset projects were run using ACDPR. The values of structural, behavioral, semantic matchings corresponding to each design pattern were

acquired. Finally, the design pattern scores are calculated using Equation 1. These scores are shown in Table I.

Table I: Scores of the Design Patterns

	Abstract Factory	Factory Method	Builder	Prototype	Singleton
P_01	1	0.29	0	0	0
P_02	1	0.29	0	0	0
P_03	1	0.29	0	0	0
P_04	1	0.29	0	0	0
P_05	0.29	1	0	0	0
P_06	0.29	1	0	0	0
P_07	0.43	1	0	0	0
P_08	0.43	1	0	0	0
P_09	0	0	1	0	0
P_10	0.14	0	1	0	0
P_11	0.43	0.29	1	0	0
P_12	0	0	1	0	0
P_13	0	0	1	0	0
P_14	0	0	0	1	0
P_15	0	0	0	1	0
P_16	0	0	0	1	0
P_17	0	0	0	1	0
P_18	0	0	0	0	1
P_19	0	0	0	0	0.33
P_20	0	0	0	0	1
P_21	0	0	0	0	1

For score $= 1$, patterns are recommended. Otherwise, patterns are suggested for threshold value 0.3. From the given recommendations, the precision and recall is measured.

Let, $tp =$ true positive, $fp =$ false positive, $fn =$ false negative. From Table I and the expected recommendations, $tp = 20,\ fp = 0,\ fn = 1$. Thus,

$$Precision = \frac{tp}{tp + fp} = \frac{20}{20 + 0} = 1$$

As, ACDPR provides no false positive recommendation, it possesses the maximum precision. And,

$$Recall = \frac{tp}{tp + fn} = \frac{20}{20 + 1} = 0.95$$

Using the precision and recall, the F-measure or the balanced F-score (F_1 score) can be calculated.

$$F_1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} = 2 \cdot \frac{1 \cdot 0.95}{1 + 0.95} = 0.97$$

For the failed case (P_19), suggestion is given to consider Singleton which was the expected recommendation. The other suggestions (P_07, P_08, P_11) have been given for Abstract Factory, which have a partial matching score of 0.43 (actual recommendations are Factory Method, Factory Method, and Builder respectively). Although these three suggestions are not applicable for the projects, these might be helpful for designers as these patterns are interrelated (referring to GoF *Related Patterns*).

IV. Conclusion

This paper introduces an approach to recommend creational design patterns using anti-patterns in the software design phase. It first derives the characteristics of the $missing$ creational patterns. Then, a tool is proposed named ACDPR, where those anti-patterns are detected in software design by structural, behavioral and semantic matching. The design patterns are recommended based on their obtained scores (for a score of 1) in these matching levels. Design patterns are also suggested based on a threshold value.

Experiments have been conducted on 21 software designs containing anti-patterns of different creational patterns. ACDPR possesses a precision of 1, recall of 0.95, and F-measure of 0.97 on this dataset. Also, for the false positive results in the recommendation (missed recall 0.05), the patterns were suggested, for a threshold value of 0.3.

Here, the characteristics of the $missing$ creational patterns are analyzed only. The future direction lies in analyzing the other $missing$ design patterns (structural, behavioral), and extending the tool to recommend those also.

Acknowledgements

This research is supported by ICT Division, Ministry of Posts, Telecommunications and Information Technology, Bangladesh. 56.00.0000.028.33.028.15-214, 24-06-2015.

References

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

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] N. Nahar and K. Sakib, "Automatic Recommendation of Software Design Patterns Using Anti-patterns in the Design Phase: A Case Study on Abstract Factory," in *Proceedings of the 3rd International Workshop on Quantitative Approaches to Software Quality (QuASoQ)*, 2015, p. 11.