# An Improved Behavioral Matching for Anti-pattern Based Abstract Factory Recommendation

Nadia Nahar*, Tarek Salah Uddin Mahmud† and Kazi Sakib‡
Institute of Information Technology, University of Dhaka, Bangladesh
*bit0327@iit.du.ac.bd, †bsse0508@iit.du.ac.bd, ‡sakib@iit.du.ac.bd

***Abstract*—For a developed project, Abstract Factory can be recommended using structural and behavioral matching between the defined anti-patterns and project code. This paper proposes a refinement of the existing behavioral matching technique, where all possible aspects of Abstract Factory behaviors are considered. This matching phase is enhanced by generating family matrix using conditional and action parse, and product type matrix using class association. Those two matrices are analysed to identify whether a pair of classes have the same product type. If no such matching is found, regeneration of family matrix by class wise parsing is done and again the matrices are analysed for recommending Abstract Factory. The approach is implemented in the form of a tool named Source code based Abstract Factory Pattern Recommender (SAFPR). The performance is justified by an experimental result analysis that showed, SAFPR is successful in Abstract Factory recommendation for all the dataset projects.**

## I. Introduction

Abstract Factory is one of the mostly used design patterns in software development. It is similar to two other design patterns, the Factory and Strategy patterns. It makes programmers confused to use this pattern perfectly which leads to the creation of anti-patterns. Interestingly, the originated anti-patterns of Abstract Factory (as well as the other design patterns) are usually defined [1], [2], [3]. To recommend the Abstract Factory from a developed source code, like other design patterns, matching both the structural (i.e. relationship between classes) and behavioral (i.e. instantiating objects and invoking methods) characteristics with those anti-patterns are required. Structural matching is relatively easier and generates false positive results due to the similarity of different design patterns' anti-pattern structures. Thus, the accuracy of recommendation depends mostly on the behavioral matching.

Perceiving the behaviors of design patterns is difficult as the intents are written in ambiguous human language. This leads to inaccurate or false assumptions on the behavioral aspects of the patterns. For correct recommendations, behaviors of the design patterns need to be obtained accurately as well as represented in a logical process to be identified in the badly designed source code. Logically defining the pattern behaviors for Abstract Factory recommendation could not been successful yet because of not considering all possible cases, i.e. different ways of instantiating families of classes such as if-else, switch-case, action-listeners, class wise etc.

One of the prominent research on design pattern recommendation is an anti-pattern based design pattern recommendation system [1]. It dynamically recommends design patterns in the software coding phase. It detects anti-patterns in the source code by structural and behavioral matching, and suggests required design patterns to mitigate those. Although the approach is very promising as it works to improve the existing software design, it is not good enough in case of Abstract Factory [4]. It assumed code segments to have conditional if-then-else or switch-case statements for instantiating families in Abstract Factory, which is not true always. Hasheminejad et. al. proposed a design pattern recommender system using a text classification approach that matches software problem descriptions with defined design pattern classes [5]. Palma et. al. proposed a question-based recommender named as Design Pattern Recommender (DPR) where designers answers to questions leaded to required design patterns using a Goal-Question-Metric (GQM) [6]. A Case Based Reasoning (CBR) approach was proposed by Gomes et. al. that used class diagrams to retrieve similar cases from knowledge base for suggesting design patterns [7]. These approaches also fail to recommend Abstract Factory correctly because of not focusing on design requirements of it.

For having accuracy in Abstract Factory recommendation, this paper proposes a refinement of the behavioral matching. At first, conditional and action listener methods parsing is done for generating family and product type matrix. Then it is decided if the Abstract Factory is appropriate to be recommended for the inputted software by analyzing the matrices through checking if the pairs of instantiated objects belong to the same families and different product types. If not, class wise parsing is performed for regenerating the family matrix. After that, the newly generated family and product type matrices are re-analysed to recommend the Abstract Factory. This behavioral phase is incorporated into a tool named as Source code based Abstract Factory Pattern Recommender (SAFPR), which is the refinement of a source based recommendation tool in the literature [1] for recommending Abstract Factory. There are two phases for the anti-pattern matching and recommendation in the proposed tool: structural and behavioral. In structural matching, project source code class structure is matched with class structures of anti-patterns using matrix matching. The structural matching phase of SAFPR is kept similar to the structural matching of the existing tool. The behavioral matching phase is replaced by the proposed refined behavioral matching.

The performance of the tool was evaluated by conducting an experimental analysis. SAFPR and the existing code-based tool [1] was implemented using Java. Seven projects were used as dataset - four of those required Abstract Factory and the other three applied Template, Strategy and Factory patterns respectively. All the project source codes were developed using Java. The result analysis showed that SAFPR provided successful recommendation in all the cases, while the existing tool was correct for five projects because of its faulty assumption.

978-1-5090-1269-5/16/$31.00 ©2016 IEEE

## II. Related Work

In the literature, several design pattern recommendation systems have been proposed. The first paper to automate the design pattern selection was proposed by Gomes et. al. even before the concept of design pattern recommendation system arrived [7]. A software Design Pattern Application (DPA) module was proposed here using CBR. It comprised three phases: retrieval of applicable DPA cases, selection of most suitable DPA case, and application of the selected DPA case. The user inputted class diagram was used for retrieving the best matched DPA case from the knowledge base. The DPA case was applied on the inputted diagram, and a new class diagram was returned. However, matching the class diagram of software cannot confirm its designing requirement for Abstract Factory, as many software class diagrams might have similar structures but not require Abstract Factory for good design.

Jebelean et. al. proposed an approach to identify $missing$ Abstract Factory in object-oriented code [3]. The approach pointed the inputted code where the classes were instantiated in undesirable manner which could be improved by application of the Abstract Factory design pattern. It identified the class/method combination of an undesirable code structure (anti-pattern) and detected that structure in inputted source codes. It computed the control paths of code using conditional statements (if-then-else, switch-case) and loops (for, while, do-while), and assumed that Abstract Factory families were instantiated in different control paths. However, there can be more control paths generated by key listeners, button listeners, class wise instantiations etc., leading the process to fail to detect $missing$ Abstract Factory in these cases.

The concept of design pattern recommender system was first brought by Guéhéneuc et. al. who proposed a simple design pattern recommendation system for recommending from the 23 GoF patterns [8]. First of all, the textual descriptions of design patterns were analyzed for extracting set of important words associated with those. Then the design patterns were recommended by comparing and ranking those using simple cosine distance between the vectors composed of those important words sets and some user selected important words. The problem is, users cannot be expected to choose the suitable important words related to the required design patterns of their software, making the recommendations erroneous. And so, Abstract Factory will only be recommended if the user choose the keywords of Abstract Factory, which is unacceptable.

Navarro et. al. proposed an approach to recommend additional design patterns while a collection of design patterns were already selected by the designer [9]. The recommendation system was based on a heuristic function to obtain the utility or rating of all the design patterns. The rating was calculated based on three factors - number of occurrences with the initially selected pattern set, types of relation with those, categorization of those. Now, as it requires an initial pattern list, it cannot be used for new software.

A question-based recommender named as Design Pattern Recommender (DPR) was proposed by Palma et. al. [6]. The approach was consisted of four steps. At first, the circumstances, in which specific design patterns were needed to be applied, were identified. Next, the circumstances were refined by identifying sub-conditions of applying those design patterns. Then, questions to be asked to the designers were identified. Finally, a Goal-Question-Metric (GQM) was formulated from those define questions. This GQM was used to lead to the required patterns from asking questions to the designers. However, the questions are generic, static and defined making it impractical to use it for all problems (software). Asking generic questions related to Abstract Factory will require the designers to have previous knowledge about it making the tool's usage critical.

Hasheminejad et. al. proposed a design pattern recommender using a text classification approach [5]. The proposed technique worked in two steps - Learning Design Patterns, and Design Pattern Retrieval. In the first step, one classifier was learnt for each pattern class (manually determined by experts). Then in the next step the given problem description was matched with the design pattern classes using text classification, and the appropriate patterns in the matched class were recommended. Problem descriptions, written in human language, are ambiguous and are not enough to identify the required design patterns. Thus, parsing problem descriptions to identify suitability of Abstract Factory can be challenging.

An anti-pattern based design pattern recommender was proposed by Smith et. al. [1]. The tool recommended design patterns dynamically during the code development. Anti-patterns residing in a code was identified through structural and behavioral matching. The design patterns required to remove the detected anti-patterns were recommended. This paper has changed the concept of design pattern recommendation as it worked for improving the existing software design by identifying the design problems (anti-patterns residing in design). The matching processes of related anti-patterns of design patterns with inputted source code has given the approach more reliability as it worked on structured information rather than the unstructured or semi-structured descriptions like the other existing approaches. However, the behavioral matching of Abstract Factory was unsuccessful due to an erroneous assumption that, the families of Abstract Factory are instantiated in if-then-else or switch-case conditions [4].

Muangon et. al. used CBR and Formal Concept Analysis (FCA) to propose a design pattern searching system [10]. This research aimed to mitigate the problem of design pattern retrieval caused by difference between author keywords and user keywords. At first, problem description was inputted by user, and a similarity function was used to retrieve similar cases for suggesting solution patterns. If the user was not satisfied, FCA was used for reformulating problem description and retrieving related cases. Then suggestions were made from those retrieved cases. However, CBR cases cannot express possible design problems, and so matching two cases cannot ensure the suitability of previously used design patterns (such as Abstract Factory) for the new case.

Another question-based design pattern advisement approach was proposed by Pavlič et. al. [11]. The advisement procedure was performed in five phases. An ontology-based design pattern repository and user's answers to questions were used in the first phase for getting a set of possible solutions as a subset of pattern containers. Then in the second phase the most suitable pattern container was selected. After that, the most suitable pattern of that selected pattern container was identified. In the next phases, the selected design pattern was verified, and the related design patterns were identified for recommendation. However, similar to the previous question-based paper [6], the static and defined design pattern related questions are not suitable for finding possible software design problems, and so the design problems related to Abstract Factory.

From the literature review, it is found that recommendation of Abstract Factory is yet an improvable field. The CBR, text matching or question-answer based approaches lack focus on software design problems that can be mitigated by using Abstract Factory. And the anti-pattern based approaches fail to consider all possible behavioral characteristics of its requirement.

## III. SAFPR: Source code based Abstract Factory Design Pattern Recommender System

Considering the above stated problems, a tool named Source code based Abstract Factory Pattern Recommender (SAFPR) is proposed. It intends to improve the software design while the source code has already been developed. For this purpose, the code pattern of the inputted project is identified by parsing source code. Code patterns can be identified by parsing two characteristics: structures and behaviors [1]. The structure defines the types of classes and relationships between those and the behavior defines how objects are created, methods are invoked and information is shared [1]. Similarly anti-patterns also have their own structural and behavioral characteristics. SAFPR stores the characteristics of the defined anti-patterns [3], [2] of the Abstract Factory pattern, and matches with the characteristics of the inputted project. SAFPR first gets the structure of the projects to match this with that of the anti-patterns of the Abstract Factory pattern. After that, the behavior of object creations in the project will be checked. In the anti-patterns of the Abstract Factory, a programmer is likely to instantiate the same families of product in different areas of the project. The families could be instantiated in if-then-else statement, switch-case statement, different action listeners (such as mouse listener, key listener) or in different classes (e.g. different single classes are instantiated in the conditional statements, and those classes instantiate the families). In behavioral matching, the instantiation of families in different areas need to be detected in inputted project. The decision of the Abstract Factory recommendation is taken after performing the behavioral matching, which is the main focus of this paper. The structural and behavioral matching approaches of SAFPR are described below.

### A. Phases of SAFPR (Structural and Behavioral Matching)

SAFPR starts with structural matching where the inputted projects structure is matched with that of the anti-patterns of the Abstract Factory. First it extracts the source code and generates a class relationship matrix of the structure. Then the matrix is matched with the anti-pattern structure matrices. The structural matching approach is similar to Smith et. al. [1].
After matching the structure, SAFPR matches the behavior of the project with the behavior of the Abstract Factory. To match the behavior, instantiation of families of objects in different areas of the project need to be detected. So, a family matrix is generated by parsing the conditional statements and the action performing methods. For checking the similarity of the family classes, product type matrix is generated. Then it is tested whether the behavior of Abstract Factory pattern is matched or not. If not, the family matrix is regenerated by class wise parsing and SAFPR rechecks whether it should recommend Abstract Factory pattern.

*1) Structural Matching:* Finding the structure of the anti-patterns in the project structure is the purpose of structural matching. Both the project and the patterns structures are defined by the set of classes and their relationships. The structures can be defined using an $n \times n$ matrix, where n is the number of classes. Usually there can be three types of relationships between classes - association, generalization and aggregation. SAFPR puts a prime number for each relations; 2 for association, 3 for generalization and 5 for aggregation which is similar to Smith et. al. [1]. For example, the matrix on Fig. 2 shows the relationships between classes in Fig. 1. In Fig. 1, class A is the parent class of B, and class C instantiates class B. So, SAFPR puts 3 in the matrix as the generalization relationship of B and A, and puts 2 for association relationship of C and B.

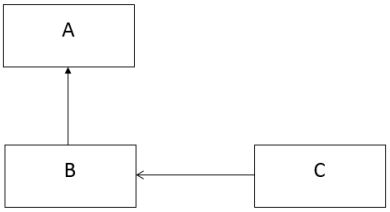


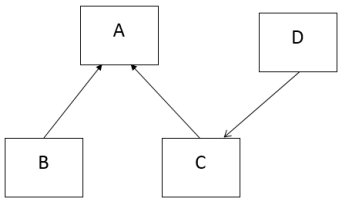

Fig. 1: Three classes and the relationships between them

| | A | B | C |
|---|---|---|---|
| A | 0 | 0 | 0 |
| B | 3 | 0 | 0 |
| C | 0 | 2 | 0 |

Fig. 2: The matrix for Fig. 1

Then SAFPR checks every node permutation in the graph of the inputted project and whether or not it matches with the anti-pattern's matrix. SAFPR finds all the subsets of classes and use their relationships to match with the anti-pattern's. Multiple types of relationships can be stored by multiplying their corresponding prime numbers. So when matching the relationship values, the project value must be divisible by the anti-pattern value. For example, if matching requires an association relationship (value 2) and the project relationship has a value 6, this is still considered a match. The graph and its representative matrix in Fig. 3 is of an anti-pattern, and likewise Fig. 4 represents an inputted project. SAFPR will find that the relationship between the set of classes {P,Q,R,S}, will create a match with the matrix in Fig. 3.

*2) Behavioral Matching:* The basic components of this process are generation of family matrix by conditional and action methods parsing, generation of product type matrix, regeneration of family matrix by class wise parsing and recommendation. The interactions of these components are shown in Fig. 5 as a form of flow chart. The discussion about these interactions are as follows.
In behavioral matching SAFPR checks if there exists at least two families of the same type objects. Using Abstract Syntax Tree (AST) parser, SAFPR first makes the family matrix by separating the objects instantiated in the if-then-else statements or switch-case statements or in the action listener methods. If two classes are instantiated with the same condition or in the same action performing method, SAFPR puts those classes in

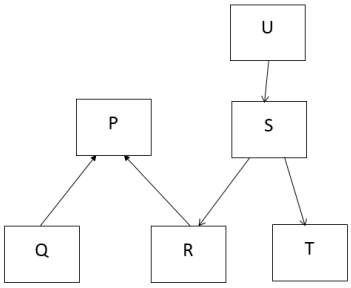



(a) Class Diagram

| | A | B | C | D |
|---|---|---|---|---|
| A | 0 | 0 | 0 | 0 |
| B | 3 | 0 | 0 | 0 |
| C | 3 | 0 | 0 | 0 |
| D | 0 | 0 | 2 | 0 |

(b) Relationship Matrix

Fig. 3: Sample Anti-pattern




(a) Class Diagram

| | P | Q | R | S | T | U |
|---|---|---|---|---|---|---|
| P | 0 | 0 | 0 | 0 | 0 | 0 |
| Q | 3 | 0 | 0 | 0 | 0 | 0 |
| R | 3 | 0 | 0 | 0 | 0 | 0 |
| S | 0 | 0 | 2 | 0 | 2 | 0 |
| T | 0 | 0 | 0 | 0 | 0 | 0 |
| U | 0 | 0 | 0 | 2 | 0 | 0 |

(b) Relationship Matrix

Fig. 4: Sample Project

the same family. The sample code of project Painter in Fig. 6 is used to describe the whole behavioral matching.

It is defined that, 1 indicates the same family and 0 indicates no relationships between the classes in the family matrix. In Paint class (Fig. 6) the objects of Circle and Red classes are instantiated in the same condition. Thus, as shown in the generated family matrix in Fig. 7, SAFPR puts 1 in between Red and Circle in the matrix as the identifier of the same family. It does the same for all classes found in the same conditional statement or action listener methods.

Using the AST parser it checks the parent classes and the child classes to identify the same type objects. The child classes of a parent class are the same products. It is defined that,

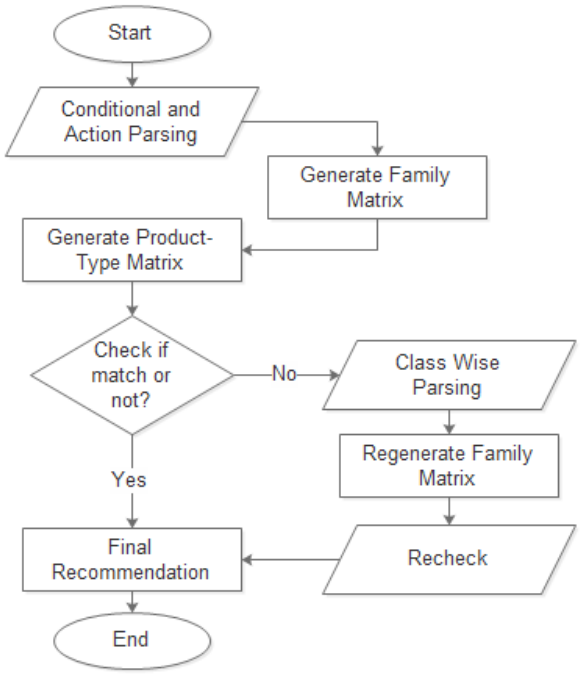



Fig. 5: Behavioral Matching Process

```
public interface IColor {
void fill();
}
public interface IShape {
void draw();
}
public class Red implements IColor{
public void fill() {
System.out.println("Filling with red");
}
}
public class Blue implements IColor{
public void fill() {
System.out.println("Filling with blue");
}
}
public class Rectangle implements IShape{
private double height=0.0;
private double width=0.0;
public Rectangle(double height, double width) {
this.height = height;
this.width = width;
}
public void draw() {
System.out.println("Rectangle is drawing");
}
}
```

```
public class Circle implements IShape{
private double r=0.0;
public Circle(double r) {
this.r=r;
}
public void draw() {
System.out.println("Circle is drawing");
}
}
public class Paint {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
int i = sc.nextInt();
IShape shape = null;
IColor c = null;
if(i==0){
shape= new Circle(2.0);
c= new Red();
}
else if(i==1){
shape= new Rectangle(2.0,3.0);
c= new Blue();
}
shape.draw();
c.fill();
}
}
```

Fig. 6: Sample Code

| | Blue | Circle | IColor | IShape | Paint | Rectangle | Red |
|---|---|---|---|---|---|---|---|
| Blue | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| Circle | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| IColor | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| IShape | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Paint | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Rectangle | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| Red | 0 | 1 | 0 | 0 | 0 | 0 | 0 |

Fig. 7: Family Matrix

1 indicates the same type and 0 indicates the different types in product type matrix. In the project in Fig. 6, Blue and

Red class have the same parent class IColor. So, those are same type products. SAFPR puts 1 between Blue and Red as the identifier of the same product type. For the code in Fig. 6, SAFPR makes the product type matrix as shown in Fig. 8. If no superclass-subclass relation is found, similarities in the names of the classes are considered for identifying same product types. However, even if the class names are not similar, the approach will fail to detect same type classes.

| | Blue | Circle | IColor | IShape | Paint | Rectangle | Red |
|---|---|---|---|---|---|---|---|
| Blue | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| Circle | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| IColor | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| IShape | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Paint | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Rectangle | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| Red | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

Fig. 8: Product Type Matrix

SAFPR makes pairs of objects that belong in the similar families but different product types using the product type (Fig. 8) and family matrices (Fig. 7). Then it tries to find if any two pairs have the same product type objects. From the family matrix of Fig. 7, SAFPR finds two pairs of same family classes. Red and Circle make a pair and Blue and Rectangle make another. To verify the families, it checks if the product type of Red and Circle matches with the other pair or not. If matches, it finds the families of objects which indicates it requires Abstract Factory. If SAFPR finds the behavioral match, it recommends the project to use Abstract Factory. If not, SAFPR regenerates the family matrix using class wise parsing. Every objects instantiated in a class is supposed to be in a family. In this family matrix, it is defined that 1 indicates the same family and 0 indicates no relationships between the classes. In the code in Fig. 6, in Paint class four objects of Blue, Circle, Rectangle and Red are instantiated. SAFPR puts 1 in all the pair of these classes. For the code in Fig. 6, SAFPR regenerates the family matrix as in Fig. 9.

| | Blue | Circle | IColor | IShape | Paint | Rectangle | Red |
|---|---|---|---|---|---|---|---|
| Blue | 0 | 1 | 0 | 0 | 0 | 1 | 1 |
| Circle | 1 | 0 | 0 | 0 | 0 | 1 | 1 |
| IColor | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| IShape | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Paint | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Rectangle | 1 | 1 | 0 | 0 | 0 | 0 | 1 |
| Red | 1 | 1 | 0 | 0 | 0 | 1 | 0 |

Fig. 9: Regenerated Family Matrix

Again SAFPR makes pair of objects of similar families, rechecks and recommends that the project should use Abstract Factory or not.

## IV. Result Analysis

The new approach is assessed by an experimental analysis on Abstract Factory. For comparative analysis, SAFPR is implemented along with the source based tool proposed by Smith et. al. [1]. GoF [12] is followed as benchmark of assessment for determining if the dataset projects require Abstract Factory or not.

### A. Environmental Setup

The tool is developed in Java programming language. The equipments used to develop the prototype are as follows.

- Eclipse Juno: Java IDE for SAFPR implementation
- AST Parser: used for static code analysis for extracting structural and behavioral information

As dataset, four projects (Car, GameScene, Painter, MazeGame) requiring the Abstract Factory pattern and three projects (Trip, MyTalkingTom and Dog) not requiring the Abstract Factory (applied Template, Strategy and Factory patterns respectively) are used. The project details are shown in Table I. The project source codes are uploaded on GitHub (https://github.com/NadiaIT/ADPR-dataset).

TABLE I: Experimented Projects

| Project Name | Number of Classes | Line of Code (LOC) |
|---|---|---|
| Car | 9 | 88 |
| GameScene | 10 | 85 |
| Painter | 9 | 99 |
| MazeGame | 12 | 395 |
| Trip | 9 | 201 |
| MyTalkingTom | 6 | 85 |
| Dog | 6 | 61 |

### B. Comparative Analysis

The projects were run using both SAFPR and the tool that was designed in [1]. The results of the experiment are demonstrated in Table II, which shows that two of the $missing$ [3] Abstract Factory patterns out of four were detected by the existing tool. On the other hand, SAFPR could detect all of those. Both the tools did not give any false positive results. The needed families in the projects and SAFPR's recommendations are shown in Table III. The "Families" column denotes the group of classes that are instantiated together and found through conditional or class wise parsing. The "Need Abstract Factory" column denotes those classes which fulfill the criteria of being Abstract Factory families as all the families have the same product types. The next column represents the families for which SAFPR recommends Abstract Factory. For example, project "Car" has three families instantiated together. However in the third family (BMW, Nitrogen), Nitrogen is not from the same product type as DieselOil and GasolineOil. So, it is not an appropriate family of the Abstract Factory.

TABLE II: Experimental Results

| Project Name | Abstract Factory Recommendation | | |
|---|---|---|---|
| | [1] | SAFPR | GoF |
| Car | Yes | Yes | Yes |
| GameScene | No | Yes | Yes |
| Painter | Yes | Yes | Yes |
| MazeGame | No | Yes | Yes |
| Trip | No | No | No |
| MyTalkingTom | No | No | No |
| Dog | No | No | No |

TABLE III: Families Requiring Abstract factory

| Project Name | Families | Need Abstract Factory | SAFPR Recommendations |
|---|---|---|---|
| Car | (BMW, DieselOil), (Ferrari, GasolineOil), (BMW, Nitrogen) | (BMW, DieselOil), (Ferrari, GasolineOil) | (BMW, DieselOil), (Ferrari, GasolineOil) |
| GameScene | (PalmTree, NarrowRiver), (CocunutTree, WideRiver) | (PalmTree, NarrowRiver), (CocunutTree, WideRiver) | (PalmTree, NarrowRiver), (CocunutTree, WideRiver) |
| Painter | (Blue, Rectangle), (Circle, Red) | (Blue, Rectangle), (Circle, Red) | (Blue, Rectangle), (Circle, Red) |
| MazeGame | (WoodenWall, FixedDoor), (GlassWall, SlidingDoor) | (WoodenWall, FixedDoor), (GlassWall, SlidingDoor) | (WoodenWall, FixedDoor), (GlassWall, SlidingDoor) |
| Trip | No families | No | No recommendation |
| MyTalkingTom | No families | No | No recommendation |
| Dog | (Poodle, Food), (Rottweiler, Food), (SiberianHusky, Food) | No | No recommendation |

The reason behind SAFPRs outperforming the result of [1] is, the existing tool assumed the Abstract Factory design pattern has only a behavioral aspect of having if-else or switch-case conditions for instantiating the families. However, families of classes can be instantiated in different classes or GUI action listeners. The existing tool misses those. On the other hand, SAFPR was successful in all cases as it checks class instantiations in if-else and switch case conditions as well as the action listeners first, then checks in all the classes and assume all objects instantiated in a class are in a family. So, it can handle the family instantiations not only in conditional statements but also in the action listeners as well as in class wise separation.

## V. Conclusion

This paper proposes a technique to Abstract Factory design pattern recommendation in the form of a tool named SAFPR. It is a refinement of the existing anti-pattern based design pattern recommendation systems.

For the recommendation, anti-patterns are identified from source code analysis in two steps - structural and behavioral matching. In structural matching, structures of the Abstract Factory anti-patterns are matched with the inputted project structure. For this, the class relationship matrix is created by source code parsing, and matrix matching is performed. The behavioral matching phase is the main contribution of this paper. The behavior of Abstract Factory anti-pattern is logically defined to be matched in four steps. At first, family matrix is generated by conditional parsing. Then the product type matrix is created from class relationship analysis. The detected families are checked to identify if those conform to be the Abstract Factory families. If not, the family matrix is regenerated by class wise parsing. And finally, the recommendation decision is taken by verifying the re-identified families. An experimental result analysis is conducted for justifying the proposed approach. The prototype of SAFPR was implemented in Java along with the tool of Smith et. al. [1]. The comparative analysis shows that SAFPR outperforms the existing tool by providing accurate recommendation for all the projects.

The future direction lies in developing recommendation systems for the other design patterns. Moreover, design diagrams can be utilized for early recommendation of design patterns before software source code development.

## Acknowledgements

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

## References

[1] 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.

[2] 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.

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

[4] S. Richardson, "Dynamically Recommending Design Patterns," Bachelor's Thesis, Stetson University, 2011.

[5] 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.

[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 IEEE International Workshop on Recommendation Systems for Software Engineering*, 2012, pp. 1–5.

[7] 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.

[8] 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.

[9] 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)*, 2010, pp. 269–270.

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

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

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