TEM Journal. Volume 10, Issue 3, Pages 1166-1174, ISSN 2217-8309, DOI: 10.18421/TEM103-21, August 2021.

Design Patterns Discovery in Source Code: Novel Technique Using Substring Match

Akshara Pande [1], Vivekanand Pant [2], Manjari Gupta [3], Alok Mishra [4,5]

[1]*Graphic Era Hill University, Bell Road, Clement Town, Dehradun, India*
[2]*IBM, IBM India Private Ltd., Gurugram, India*
[3]*(Computer Science), DST-CIMS, Institute of Science, BHU, Varanasi, India*
[4]*Faculty of Logistics, Molde University College (Specialized University in Logistics), 6410 Molde, Norway*
[5]*Department of Software Engineering, Atilim University, Ankara 06830, Turkey*

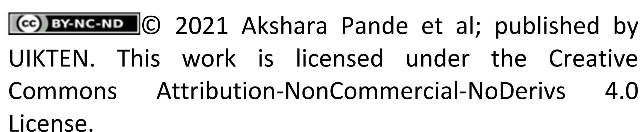
Abstract **– The role of design pattern mining is a very significant strategy of re-engineering as with the help of detection one could easily understand complex systems. Of course, identifying a design pattern is not always a simple task. Additionally, pattern recovering methods often encounter problems dealing with space outburst for extensive systems. This paper introduces a new way to discover a design pattern based on an Impact Analysis matrix followed by substring match. UML diagrams corresponding to codes are created using Visual Paradigm Enterprise. Impact Analysis matrices of these UML diagrams are converted to string format. Considering system code string as main string and design pattern string as a substring, the main string is further decomposed. A substring match technique is developed here to discover design patterns in the source code. Overall, this procedure has the potential to convert the representation of system design and design pattern in ingenious shapes. In addition, this method has the advantage of moderation in the size. Therefore, this approach is beneficial for Software professionals and researchers due to its simplicity.**

Keywords **– Design patterns, UML diagrams, Visual Paradigm Enterprise software, Impact Analysis Matrix, Substring match.**

DOI: 10.18421/TEM103-21
https://doi.org/10.18421/TEM103-21

Corresponding author: Manjari Gupta,
(Computer Science), DST-CIMS, Institute of Science, BHU, Varanasi, India.
Email: manjari@bhu.ac.in

Received: 23 May 2021.
Revised: 29 June 2021.
Accepted: 09 July 2021.
Published: 27 August 2021.


© 2021 Akshara Pande et al; published by UIKTEN. This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 4.0 License.

The article is published with Open Access at www.temjournal.com

1. Introduction

A design pattern is a general recurring explanation for commonly encountered issues in software design. A design pattern being not a final solution cannot be directly converted to code but provides a direction of solving an issue and is reusable in many different situations. Since software developers tend to unravel many similar sorts of problems, it is advisable to integrate similar elements to other methods in any software solution. It is an illustration or format providing ways of tackling an issue that is useful in a wide range of circumstances [1] and reduces the technical risk of projects by avoiding the use of new, unconventional designs. Design patterns are well written and understood by software developers, so that the application is understood in certain terms and the final product will have better comprehension value. If the solution is easy to understand, the extension will be easier to maintain. When considering the development pipeline of a project, development time gradually decreases as awareness of design pattern increases. Therefore, the field of design patterns attract researchers from academia and industry both.

In the present study, we will focus on the issue of identifying the design patterns, more specifically whether a specific design pattern is present in source code or not, with the help of Impact Analysis Matrix followed by substring match. We use the matrix-based approach to simplify and speed up the design pattern identification process.

1.1. Problem Statement

Researchers developed different tools and techniques for automatically detecting design patterns from source code. It has been seen that these tools and techniques do not agree on their results if these are applied to the same case study. There are many reasons for this: 1) They generally use different specifications of design patterns as there is no formal

definition of design patterns; 2) Generally, tools are developed by focusing on some particular (one/more) design patterns.

In this paper we propose a technique that is applicable on detecting any design pattern. In this paper, we describe our technique by using two design patterns: factory method and proxy. However, the same way it can be applied to detect any other design pattern.

1.2. Objectives and Research Questions

The main motivation to identify design patterns is to provide a good perspective to understand the original design decisions. This is also beneficial in various areas of software engineering such as re-engineering, refactoring, maintenance, evaluating software quality, understanding programs, and enhancing software documentation.

This section presents research questions that will be answered in this paper:

RQ1: Does the change in representation of system design and design pattern make the process of design pattern detection easy?
Motivation: To study and analyze the effectiveness of string format representation of design patterns on the results of their detection process.
RQ2: Are we able to find design pattern occurrence if all the relationships are considered altogether?
Motivation: To study the impact of one/all relationships on the results of discovering design patterns process.

1.3. Contributions

In summary, the main factors of proposed method for discovering design patterns are as follows:

1) In this paper we are able to find design pattern existence with the help of matching between two substrings, one corresponding to system design and other corresponding to design pattern. The matching could be complete matching or partial matching based on relationships. If all relationships present in design pattern are there in system design too, this will belong to complete match. But if only one or few relationships exist in design patterns that are there in system design, then it will be a partial match case.
2) We have implemented our code in python and python is portable. So, there is no dependency on the platform as to run the same code, and we do not have to make any change according to the operating system.
3) This method provides decomposition of system design. First a size of relationship matrix of a particular design pattern is considered. Assuming the size of design pattern as a window, traversing of window is done for the same relationship in the system under consideration.
4) Simple: Once we have a set of decomposed substrings corresponding to system design, the task of design pattern detection becomes very easy with the help of the membership operator ("in") of python.

The paper is organized as follows. In section 2, we discuss the background study of the paper. In section 3 we present related works. Research methodology and our proposed design patterns detection approach is explained in section 4. Section 5 shows experimental results and discussion. Finally, we conclude and give future research scope in section 6.

2. Background

2.1. Design Patterns

Design patterns are general solutions provided by expert designers to common design problems. Twenty-three design patterns were given by four authors Erich Gamma, Richard Helm, Ralph Johnson and John Vlisside in the year 1994 [1]. Design patterns are the best evolved solutions to common problems of software development solved by expert designers over a long period of time. Therefore, these are helpful to novice developers as they directly use these solutions without wasting their time to reinvent the solution again and again. Since design patterns have been reused several times, we can say these are time tested and thus software development using design patterns improves quality of product as well as productivity of developers. As we know business requirements demand short delivery time for software, design patterns can help us in achieving this goal also. Therefore, design patterns help in achieving all three basic drivers cost, quality and productivity in a software product.

2.2. Visual Paradigm Enterprise

Visual Paradigm is a tool that supports SysML, BPMN (Business Process Modelling Notation) and UML2. It also includes report production and code engineering features. It also contains the feature of reverse engineering from code. Besides this, Entity Relationship Diagrams (ERD) and Object Relational Mapping Diagrams (ORM) are both also supported by Visual Paradigm. It can also be used to identify the relationships among model elements with the use of a matrix. Two enhancements are made in the latest version of visual paradigm: i) supported multiple relationship types; ii) supported showing elements' user ID in matrix's column and row header. (https://en.wikipedia.org/wiki/Visual_Paradigm).

2.3. UML Diagram Corresponding to Source Code and Design Patterns

We first selected Java source code for system design and for design patterns and did the reverse engineering to convert code into UML diagrams with the help of Visual Paradigm Enterprise (https://www.visual-paradigm.com/). The UML diagram of source code (https://www.javatpoint.com/abstract-factory-pattern) is illustrated in Figure 2. This UML diagram consists of two relationships Generalization and Realization respectively. The corresponding matrix of UML diagrams can be generated with the help of the Impact Analysis feature of Visual Paradigm Enterprise software.

We selected two design patterns, factory method and proxy design patterns for present study. However, as we mentioned above our technique can be used to detect instances of any design pattern. In the factory method design pattern, it is possible for subclasses to create objects. Factory method design pattern makes design flexible and less complicated. Abstract factory design pattern is somewhat similar to factory method but Abstract factory has family prominence. The next one is the Proxy design pattern which is structural and allows us to allocate an alternative for another object. We are allowed to perform something prior or post request is sent to the original object. Factory method and proxy design pattern corresponding UML diagrams generated are shown in Figure 3(a) and Figure 3(b) respectively. Factory method design pattern only consists of Generalization relationship whereas Proxy design pattern consists of two relationships: Realization and Association. Impact Analysis Matrix of source code as well as of design pattern can be downloaded in comma separated values (csv) format. Impact Analysis Matrix contains the information about relationships present in the code.

3. Related Work

Gamma et al. also famously called “Gang of Four (GoF)” suggested classic formats for their 23 design patterns [1]. Design pattern has continued to evolve since the release of the GoF book, primarily because software developers face new challenges related to hardware changes and requirements. Now-a-days Design patterns are divided into four categories: creational, structural, behavioral, and concurrency. Using classes and object interactions, design patterns are utilized to advance software maintenance. The information about design patterns utilized in source code assists programmers to better understand the design. During the re-engineering, identification of design patterns is significant in order to get essential knowledge. In the past, several design pattern mining techniques have already been designed. One of the reports put forth an attempt towards the automatic identification of design patterns with the help of reverse engineering of small-talk code [2]. A couple of advances of design pattern mining focus upon formal specification such as the essential role for composition and formalization of design patterns [3], [4]. In one of the research, database inquiries were done to recoup the events of design patterns [5].

In the most recent decade, graph theory has a huge assortment of uses to explain numerous functional and genuine applications. One of the significant applications of graph theory is in the area of design pattern detection. Some of the graph matching methods are applied to compute the closeness of two vertices [6]. In one of the reports, mining of design patterns were done using similarity scoring algorithms [7]. Nonetheless the fundamental downside of this algorithm is that it was not capable of matching the similarity of two graphs. To overcome this issue, a research [8] suggested another strategy of Template Matching which ascertains the similitude between subgraphs.

In some of the design pattern detection techniques UML diagrams are likewise utilized [9], [10]. In recent years, Machine learning is also being used in design patterns detection and similar problems [11], [12].

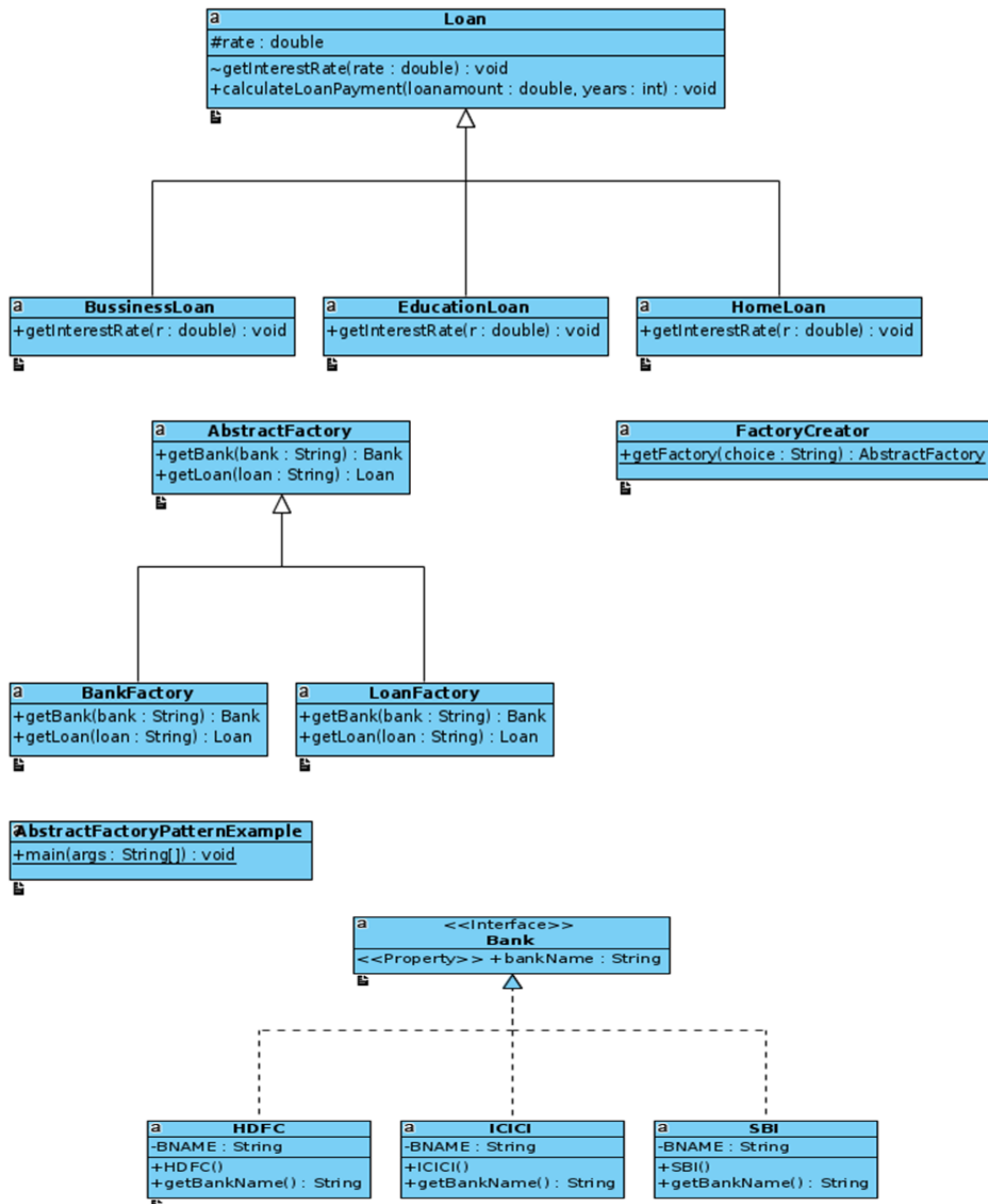
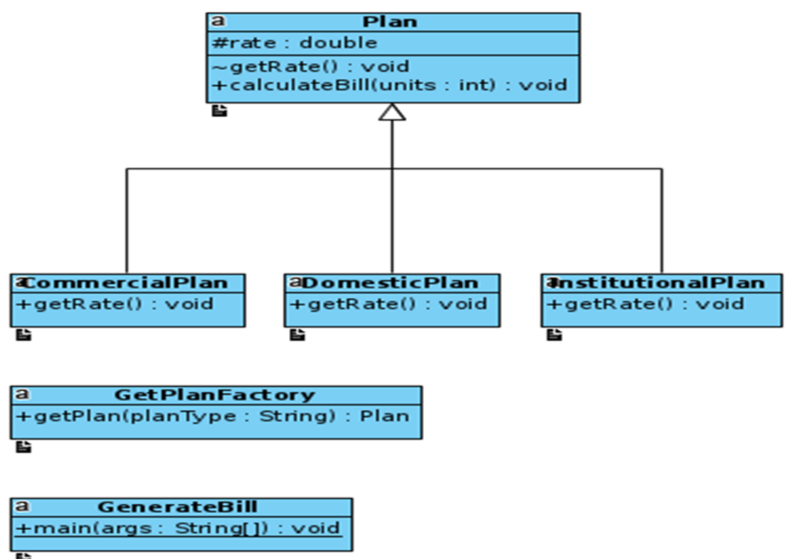
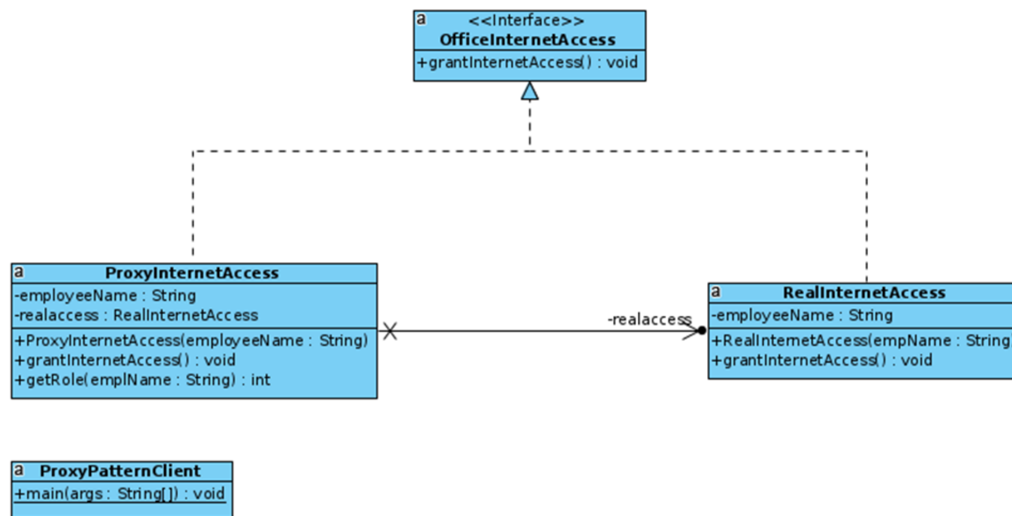


Figure 1. UML Diagram generated by reverse-engineering of Java source code with the help of Visual Paradigm Enterprise software (https://www.visual-paradigm.com/)



(a) Factory method design pattern



(b) Proxy design pattern

Figure 2. UML Diagrams of design patterns generated by reverse-engineering of Java source code with the help of Visual Paradigm Enterprise software (https://www.visual-paradigm.com/)

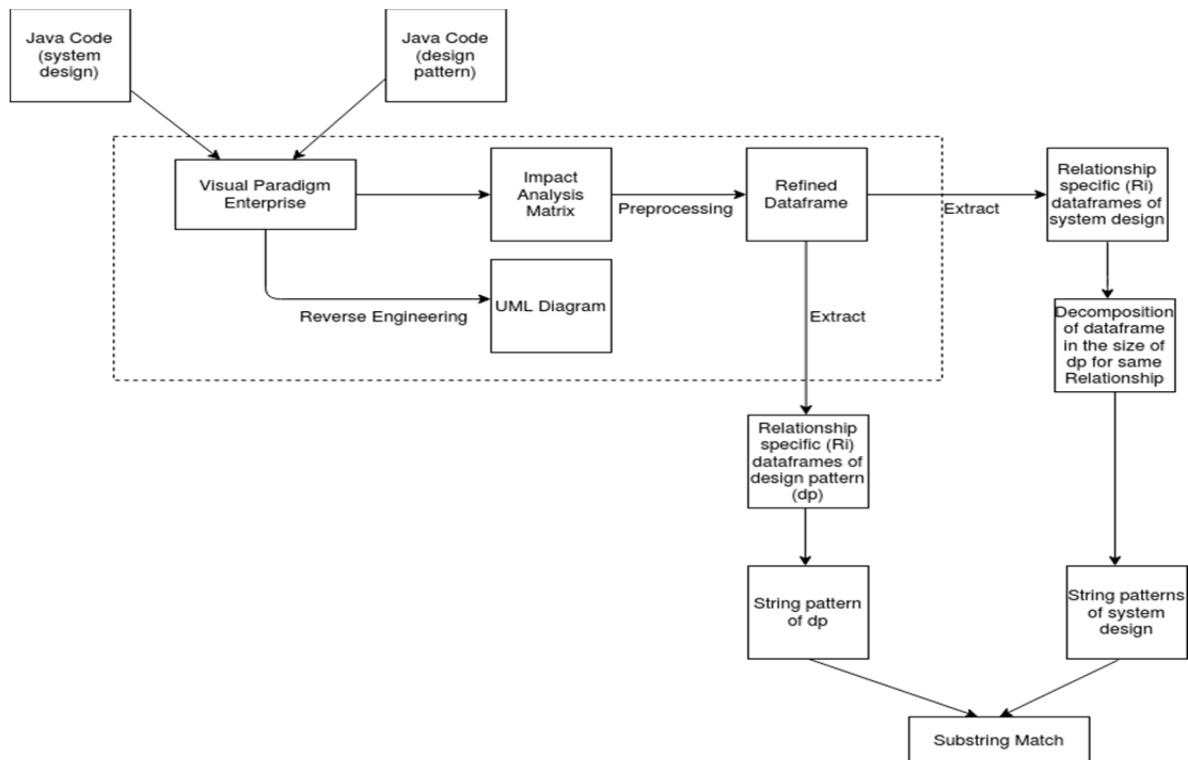


Figure 3. Steps of the methodology

4. Research Methodology and the Proposed Approach

The methodology consists of several steps. These are the processing steps basically starting from code to pattern generation. It is very difficult to understand the presence of any design pattern from the code directly. UML diagrams are simpler and quicker to understand than textual content. So, these diagrams are better suited than a huge number of lines of code description. Therefore, we converted system design and design pattern Java code into UML diagrams with the help of Visual Paradigm Enterprise.

Visual Paradigm Enterprise also provides an automatically formed matrix called Impact Analysis Matrix. This matrix helps to determine the connectivity and relationships between elements. But this matrix may contain many null values as well so preprocessing is required to handle them and after preprocessing, we get the refined matrices.

Further refined matrices may be extracted for particular relationships. The relationship matrix of refined system design is further decomposed in the

size of design pattern. String patterns are generated for both system design and design patterns from the refined relationship matrices. Then substring match is applied to find the occurrence of design pattern in system design. The block diagram, presented in Figure 3, depicts the overall steps involved. We also discussed each step-in detail in this section.

4.1. Pre-processing of Impact Analysis Matrix

We used a jupyter notebook (Anaconda Navigator) to read csv files corresponding to Impact Analysis Matrices of source code and design pattern. We used panda's dataframe to read the csv file. The 'NaN' values present in the dataframe were replaced by zero as they indicate absence of any relationship between two classes. Each class is treated as nodes of the graph. To denote a relationship between two classes, we used specific values in the dataframe. For instance, we assumed '1' for presence of Generalization relationship, '2' for Realization relationship and '3' for Association relationship. Supplementary Table 1 denotes Impact Analysis dataframe for system design with relationships Generalization and Realization. However, Table 2 and Table 3 denote Impact Analysis data frames for factory method and proxy design patterns with their relationships respectively.

Table 1. Impact Analysis dataframe for system design after pre-processing

(11) Class\ (11) Class	Abstract Factory	Bank	BankFactory	Bussiness Loan	EducationLoan	HDFC	Home Loan	ICICI	Loan	Loan Factory	SBI
Abstract Factory	0	0	0	0	0	0	0	0	0	0	0
Bank	0	0	0	0	0	0	0	0	0	0	0
BankFactory	1	0	0	0	0	0	0	0	0	0	0
BussinesLoan	0	0	0	0	0	0	0	0	1	0	0
EducationLoan	0	0	0	0	0	0	0	0	1	0	0
HDFC	0	2	0	0	0	0	0	0	0	0	0
HomeLoan	0	0	0	0	0	0	0	0	1	0	0
ICICI	0	2	0	0	0	0	0	0	0	0	0
Loan	0	0	0	0	0	0	0	0	0	0	0
LoanFactory	1	0	0	0	0	0	0	0	0	0	0
SBI	0	2	0	0	0	0	0	0	0	0	0

Table 2. Impact Analysis dataframe for factory method design pattern after pre-processing

(4) Class \ (4) Class	CommercialPlan	DomesticPlan	InstitutionalPlan	Plan
CommercialPlan	0	0	0	1
DomesticPlan	0	0	0	1
InstitutionalPlan	0	0	0	1
Plan	0	0	0	0

Table 3. Impact Analysis dataframe for proxy design pattern after pre-processing

(3) Class \ (3) Class	OfficeInternetAccess	ProxyInternetAccess	RealInternetAccess
OfficeInternetAccess	0	0	0
ProxyInternetAccess	2	0	0
RealInternetAccess	2	3	0

4.2. Relationship-wise extraction of Dataframe

Particular relationship specific data frames were generated. In the current scenario, system design consists of two relationships hence two data frames were generated: one for Generalization (Table 4) and other for Realization (Table 5). Likewise, data frames were generated for design patterns as well. Factory method design pattern consists of only one relationship i.e. Generalization so there is no need to extract that. But the proxy design pattern consists of two relationships - Realization (Table 6) and Association (Table 7)) hence two data frames were generated for that.

Table 4. Generalization relationship dataframe for system design

(11) Class\ (11) Class	AbstractFactory	Bank	BankFactory	BussinessLoan	EducationLoan	HomeLoan	Loan	LoanFactory
AbstractFactory	0	0	0	0	0	0	0	0
Bank	0	0	0	0	0	0	0	0
BankFactory	1	0	0	0	0	0	0	0
BussinessLoan	0	0	0	0	0	0	1	0
EducationLoan	0	0	0	0	0	0	1	0
HomeLoan	0	0	0	0	0	0	1	0
Loan	0	0	0	0	0	0	0	0
LoanFactory	1	0	0	0	0	0	0	0

Table 5. Realization relationship dataframe for system design

(11) Class \ (11) Class	AbstractFactory	Bank	HDFC	ICICI	Loan	SBI
AbstractFactory	0	0	0	0	0	0
Bank	0	0	0	0	0	0
HDFC	0	2	0	0	0	0
ICICI	0	2	0	0	0	0
Loan	0	0	0	0	0	0
SBI	0	2	0	0	0	0

Table 6. Realization relationship dataframe for proxy design pattern

(3) Class \ (3) Class	OfficeInternetAccess	ProxyInternetAccess	RealInternetAccess
OfficeInternetAccess	0	0	0
ProxyInternetAccess	2	0	0
RealInternetAccess	2	0	0

Table 7. Association relationship dataframe for proxy design pattern

(3) Class \ (3) Class	OfficeInternetAccess	ProxyInternetAccess	RealInternetAccess
OfficeInternetAccess	0	0	0
ProxyInternetAccess	0	0	0
RealInternetAccess	0	3	0

4.3. Decomposition of Relationship dataframe of System Design

The relationship Ri of system design dataframe was taken into account and decomposed further in the size of relationship Ri of design pattern dataframe. The relationship Ri of design pattern dataframe acted like a window which would scroll over the relationship Ri of system design dataframe. For the generalization relationship of factory method design pattern (size 4X4), we get 25 data frames corresponding to the generalization relationship of system design (Table 8).

Table 8. Illustration of decomposition of system design for generalization relationship

(11) Class \ (11) Class	AbstractFactory	Bank	BankFactory	BussinessLoan	EducationLoan	HomeLoan	Loan	LoanFactory
AbstractFactory	0	0	0	0	0	0	0	0
Bank	0	0	0	0	0	0	0	0
BankFactory	1	0	0	0	0	0	0	0
BussinessLoan	0	0	0	0	0	0	1	0
EducationLoan	0	0	0	0	0	0	1	0
HomeLoan	0	0	0	0	0	0	1	0
Loan	0	0	0	0	0	0	0	0
LoanFactory	1	0	0	0	0	0	0	0

4.4. *Pattern Formation*

All the relationship data frames have only two values: '0' indicates absence of relationship and 'k' indicates presence of relationship 'k' between two classes. The string consisting of 0 and k can be generated corresponding to the dataframe. Here, we generated strings for relationship dataframes of decomposed system design and design pattern.

4.5. *Substring Match Algorithm*

Substring match algorithm actually consists of three main steps:

a. Decomposition of relationship dataframe of system design;
b. String generation for relationship dataframes of system design and design pattern;
c. Substring match.

4.5.1. Algorithm: decomp_relationship_source(Si,DPi)

1. Determine the size of 'i' relationship dataframe of design pattern DP (n=len(DPi)
2. Determine the size of 'i' relationship dataframe of Source code S (m=len(Si)
3. Find all n*n dataframes in m*m dataframe :decomp(m,n)
4. decomp(Si,m,n)

   4.1. for each x extract entries upto m-n+1 in rows of dataframe Si

   4.2. for each y extract entries upto m-n+1 in columns of dataframe Si

   4.3. Obtained all the dataframes of size n*n; number of dataframes=(m-n+1)*(m-n+1)

4.5.2. Algorithm: String_gen(df)

1. Determine the size of dataframe df (k=len(df))
2. Assume presence of relationship between two classes as Relationship_status = p
3. Absence of relationship between two classes is Relationship_status = 0
4. For p each 0th row to kth row store relationship_status

4.5.3. Substring_match(Si,DPi)

1. Call decomp_relationship_source(Si,DPi); this function will return an array of decomposed dataframes (name it as arr_decomp_Si) of source code Si for relationship i
2. Call String_gen(DPi); this function will generate string for dataframe of DP for relationship i
3. Call String_gen(arr_decomp_Si); this function will generate strings for set of dataframes of arr_decomp_S for relationship i
4. Match whether string of DPi is a subset of set of strings of arr_decomp_Si

5. Experimental Results and Discussion

In this section, we would explain the Substring_match algorithm with the help of factory method and proxy design patterns. The purpose is to find occurrence of design patterns in provided system design.

5.1. *Detecting Factory Method design pattern instances in System Design*

After preprocessing of dataframes of design pattern and system design, relationship-wise different dataframes were extracted. Factory method design pattern has only one relationship generalization (Table 2). The main source code has two relationships: generalization and realization, hence two dataframes corresponding to each relationship were obtained (Table 4 and Table 5). The objective was to find relationship generalization of dataframe of design pattern in the dataframe of same relationship for system design. System design was further decomposed based on the size of generalization dataframe of design pattern and we obtained 25 sets of system design (Figure SF1 (Supplementary Figure 1)).

Next was to generate strings for generalization relationship for design pattern and decomposed system design. Generalization relationship of factory method design pattern formed string: '0001000100010000'. However, generalization relationship of decomposed system design formed 25 strings as:

'0000000010000000','0000000000000000','0000000000000000','0000000000000001','0000000000000010','0000100000000000','0000000000000000','0000000000000000','0000000000010001','0000000000100010','1000000000000000','0000000000000000','0000000000000000','0000000100010001','0000001000100010','0000000000000000','0000000000000000','0000000000000000','0001000100010000','0010001000100000','0000000000001000','0000000000000000','0000000000000000','0001000100000000','0010001000000000'

Next objective was to find the occurrence of a string of design patterns in strings of a set of system designs for the same relationship. We found that factory method design patterns are present in system design for generalization relationships (Figure 4).

```
if s1 in arr1 :
    print("Output: Design Pattern detection is successful")
else :
    print("Output: Couldn't find the given Design Pattern, Please try another"
Output: Design Pattern detection is successful
```

Figure 4. Screenshot of output for searching occurrence of string of factory method design pattern for generalization relationship (s1) in sets of strings for system design for generalization relationship (arr1) of system design

5.2. *Find presence of Proxy design Pattern in System Design*

Proxy design pattern has two relationships realization and association namely, presented in Table 6 and 7 respectively. Our objective was to establish if a proxy design pattern is present in system design or not. System design realization relationship was further decomposed based on the size of dataframe of design pattern and we obtained 16 sets of system design (Figure SF2 (Supplementary Figure 2)).

Next was to generate strings for design pattern and decomposed system design. Proxy design pattern formed string: '000200230'

However, realization relationship of decomposed system design formed 16 strings as: '000000020','000000200','000000000','000000000','000020020','000200200','000000000','000000000','020020000','200200000','000000000','000000000','020000020','200000200','000000000','000000000'

Next objective was to find occurrence of string of design pattern in strings of set of system design. We found that proxy design pattern does not exist in system design, presented in Figure 5.

```
if s2 in arr3 :
    print("Output: Design Pattern detection is successful")
else :
    print("Output: Couldn't find the given Design Pattern")
Output: Couldn't find the given Design Pattern
```

Figure 5. Screenshot of output for searching occurrence of string of proxy design pattern (s2) in sets of strings for system design for realization relationship (arr3) of system design

However, if we drop association relationships and consider only the realization relationship of the proxy design pattern, in that case the formed string is: '000200200'. And we found out a proxy design pattern is present in system design for the same relationship realization, shown in Figure 6.

```
if s3 in arr3 :
    print("Design Pattern detection is successful")
else :
    print("Couldn't find the given Design Pattern, Please try another")
Design Pattern detection is successful
```

Figure 6. Screenshot of output for searching occurrence of string of proxy design pattern for realization relationship (s3) in sets of strings for system design for realization relationship (arr3) of system design

Analysis of Results

RQ1: The answer to the first research question we can provide as follows:
Substring match algorithm (section 4.5) is a simple method which contains two important functions:

i. decomp_relationship_source() - it allows us to decompose the system design () in the size of the design pattern.
ii. String_gen() - Using this function, we are able to convert matrices of decomposed system design and design patterns into strings. It is pretty much obvious from above results that once we obtain strings corresponding to system design and design pattern we can easily perform matching between both.

RQ2: From the above result it is clear that we have to consider one relationship at a time. In the case of Proxy Design Pattern (section 5.2), two relationships realization and association are considered altogether. But in system design only the realization relationship is present. Thus, substring match algorithm results non-existence of proxy design pattern. But when we drop association relationship and consider realization relationship, the same substring match algorithm shows the presence of proxy design pattern in system design.

6. Conclusion and Future Research

Common solutions exist in software design patterns for fixing troubles in the massive software systems. They play important roles in forward and reverse engineering both. Design pattern detection enhances understandability and quality. We hope that our work will be beneficial for software professionals and researchers. One of the advantages of the method is to identify design patterns based on relationships. The second advantage of our method is simple decomposition of system design matrix in the length of design pattern. It makes the process of presenting system design in terms of substrings easier. This paper gives the insights of the presence of three kinds of design pattern detection using substring match: complete match, partial match and no match. For instance, factory method design pattern is completely detected in source design as this design pattern

contains only one relationship: generalization. In the case of proxy design pattern, out of two only one relationship (realization) was found in system design, this indicates partial match. If none of design pattern relationships is found in system design, this will be the case of no match.

In the present paper, we use Visual Paradigm Enterprise software to generate Impact Analysis Matrix. In future we will further extend this work and focus on developing a tool which will be able to generate matrices by itself so that users need not to run around to perform different tasks. Thus, our future tool will be more user friendly to detect design patterns. The limitation of the study is that we are not able to include more than one relationship for design pattern detection. In future, we will try to improve the method so that we can take all the relationships altogether, and our algorithm will be able to provide relationship status specifically for both presence and absence.

References

[1]. Gamma, E., Helm, R., Johnson, R., Vlissides, J., & Patterns, D. (1995). *Elements of reusable object-oriented software* (Vol. 99). Reading, Massachusetts: Addison-Wesley.

[2]. Brown, K. (1996). *Design reverse-engineering and automated design-pattern detection in Smalltalk*. North Carolina State University. Dept. of Computer Science.

[3]. Wang, Y., & Huang, J. (2008). Formal modeling and specification of design patterns using RTPA. *International Journal of Cognitive Informatics and Natural Intelligence (IJCINI)*, *2*(1), 100-111. https://doi.org/10.4018/978-1-60566-902-1.ch013

[4]. Alnusair, A., Zhao, T., & Yan, G. (2014). Rule-based detection of design patterns in program code. *International Journal on Software Tools for Technology Transfer*, *16*(3), 315-334. https://doi.org/10.1007/s10009-013-0292-z.

[5]. Rasool, G., Philippow, I., & Mäder, P. (2010). Design pattern recovery based on annotations. *Advances in Engineering Software*, *41*(4), 519-526. https://doi.org/10.1016/j.advengsoft.2009.10.014

[6]. Blondel, V. D., Gajardo, A., Heymans, M., Senellart, P., & Van Dooren, P. (2004). A measure of similarity between graph vertices: Applications to synonym extraction and web searching. *SIAM review*, *46*(4), 647-666. https://doi.org/10.1137/s0036144502415960

[7]. Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., & Halkidis, S. T. (2006). Design pattern detection using similarity scoring. *IEEE transactions on software engineering*, *32*(11), 896-909. https://doi.org/10.1109/tse.2006.112.

[8]. Dong, J., Sun, Y., & Zhao, Y. (2008, March). Design pattern detection by template matching. In *Proceedings of the 2008 ACM symposium on Applied computing* (pp. 765-769).

[9]. Bergenti, F., & Poggi, A. (2002). Improving UML designs using automatic design pattern detection. In Handbook of Software Engineering and Knowledge Engineering: Volume II: Emerging Technologies (pp. 771-784).

[10]. Wenzel, S., & Kelter, U. (2006, October). Model-driven design pattern detection using difference calculation. In *Workshop on Pattern Detection for Reverse Engineering*.

[11]. Mhawish, M. Y., & Gupta, M. (2020). Software Metrics and tree-based machine learning algorithms for distinguishing and detecting similar structure design patterns. *SN Applied Sciences*, *2*(1), 1-10.

[12]. Mhawish, M. Y., & Gupta, M. (2020). Predicting Code Smells and Analysis of Predictions: Using Machine Learning Techniques and Software Metrics. *Journal of Computer Science and Technology*, *35*(6), 1428-1445.