Visualizing Composition in Design Patterns

Zaigham Mushtaq, Kiran Iqbal, Ghulam Rasool
COMSATS Institute of Information Technology, Defence Road, Lahore, Pakistan

***Abstract*—Visualization of design patterns information play a vital role in analysis, design and comprehension of software applications. Different representations of design patterns have been proposed in literature, but each representation has its strengths and limitations. State of the art design pattern visualization approaches are unable to capture all the aspects of design pattern visualization which is important for the comprehension of any software application e.g., the role that a class, attribute and operation play in a design pattern. Additionally, there exist multiple instances of a design pattern and different types of overlapping in the design of different systems. Visualization of overlapping and composition in design patterns is important for forward and reverse engineering domains. The focus of this paper is to analyze the characteristics, strengths and limitations of key design pattern representations used for visualization and propose a hybrid approach which incorporates best features of existing approaches while suppressing their limitations. The approach extends features which are important for visualizing different types of overlapping in design patterns. Stereotypes, tagged values, semantics and constraints are defined to represent the design pattern information related to attributes and/or operations of a class. A prototyping tool named VisCDP is developed to demonstrate and evaluate our proposed.**

Keywords—Design patterns; Visualization; Program Comprehension; Reverse engineering; Composition

I. Introduction

Design patterns are proven solutions and they are composed with each other for the development of software applications [1, 2, 3]. The composition of design patterns [4, 5] when applied in an effective manner solves many generic and specific problems related to any object oriented programming domain. The visualization of pattern related information in UML diagrams and visualization of overlapping in recognized design patterns plays an important role for the program comprehension during forward as well as reverse engineering. The importance of composite visualization of design patterns is also highlighted by authors [32, 33, 34]. Mostly, design patterns are modeled using visual and formal languages such as UML [6], DPML [23], LePUS [26], RSL [31] etc. UML is a semiformal type of modeling language which is widely accepted by the academia and industry. It has a collection of visual notation techniques to build, specify, visualize, modify and document the visual models of software systems.

It is realized by Dong et al. [7] that standard UML is unable to keep track of the roles each modeling element plays in a design pattern and other design pattern related information. Therefore, some authors presented extended UML based design pattern specification and visualization approaches [3, 9, 13, 27, 28, 30]. The alternate visual and/or textual notations are also proposed to visualize the pattern related information in the software design. Porras et al. [8] concluded that all existing approaches are not capable to include all design pattern related information that is important for the comprehension of a software design. Therefore, it is important to carefully investigate all the notations. Authors in [8] presented a framework to compare the current and future notations based on participation, role and compositions of artifacts which play key roles in designing and composition of design patterns. They realized that different notations have their strengths and limitations. The limitations of existing notations provide opportunities to researchers for devise new notations that would further overcome identified limitations while combining the best features of current notations. We critically analyze state of the art design pattern representation approaches in the literature review Section 2.

The selection of an appropriate notation is important for designers, maintainers and reverse engineers. We selected Pattern: Role notation [3] and stereotype enhanced UML diagrams presented by Dong et al. [9] as baselines to propose our hybrid approach. These two notations are most representative and they are also used by other researchers [8]. Pattern: Role Notation is highly readable and informative, but it cannot represent the roles that an operation and attribute plays in a design pattern. This notation also cannot distinguish the multiple instances of the same design pattern. Stereotype enhanced UML diagrams are defined mainly by presenting a new UML profile for the representation and the visualization of the design patterns in their composed form. This approach represents the role each modeling element plays in a design pattern, but it is strongly textual thus text overload can considerably increase the size of the classes as well as make the classes harder to read. Furthermore, both these approaches do not focus on the visualization of different types of overlapping in design patterns which are important for the comprehension of software applications. When different roles (patterns, classes, operations, attributes) of a design pattern are reused in other patterns in the same application design, we call these roles overlapping. Such overlapping are very common in software applications as discussed by [4, 15, 16, 24, 25]. While analyzing results of different design pattern recovery tools, we recognized that proven composite patterns are present with different overlapping. For example, the Java AWT framework is composed of different patterns. Similarly, we found overlapping in different roles of Abstract Factory pattern and visitor pattern as mentioned in well know book by Gamma et al. [35]. Overlapping in design patterns give information about the level of coupling in different patterns and their roles. The detection and visualization of overlapped pattern roles are important for maintenance, comprehension and change impact analysis.

In order to overcome the limitations of design pattern visualization techniques [3, 9], this paper is intended to propose an approach that integrates the best features of Pattern: Role notation [3] and stereotype enhanced UML diagrams [9] while surpassing their limitations. Our approach extends new features for highlighting different types of overlapping and it is equally beneficial for forward and reverse engineering activities. The presented approach helps to visualize the following pattern related information in the recognized design patterns as contribution of this work:

- To visualize the role that a class, attribute and operation play in a design pattern;
- To visualize the multiple instances of a design pattern in class diagrams;
- To visualize One to one, one to many and many to many overlapping in design patterns;
- A proposed hybrid approach supplemented with prototyping tool VisCDP to support above mentioned claims;
- Evaluation and comparison of approach on a case study.

The rest of paper is structured as follows: We discuss state of the art on design patterns visualization approaches in Section II. Section III presents the comparison of existing approaches based on different attributes. A proposed hybrid approach used for visualization of composition in design patterns is laid down in Section IV. The prototyping tool used to validate concept of the proposed approach is discussed in Section V. Section VI discusses evaluation of approach with the help of a case study. Finally, we conclude and sketch future directions in Section VII.

II. Related Work

Design patterns are widely used in open source and industrial applications as they are known solutions to recurring problems [10]. Generally, the software developers come across with the certain kinds of problems repeatedly in their daily routines. They evaluate such problems and their context by referring to some existing design patterns and select a particular design pattern based on their needs and requirements. The reuse of a design pattern helps the software architect to reuse the knowledge that has already been documented and tested in order to improve the quality of their products. The visualization of design patterns information in large and complex software systems is decisive for the comprehension of software applications [21]. The better visualization of design patterns information enables better comprehension of examined applications [29]. Different authors proposed different approaches for the visualization of design patterns in software designs which are discussed as follows:

Smith [11] proposed a hierarchical approach called Pattern Instance Notation (PIN) to visually represent the composition of patterns. This notation provides a simple visualization approach which is not only for design patterns, but it is also for other abstractions of software engineering. A suitable graphical notation based on boxes and lines is devised in this approach, defining three modes namely collapsed, standard and expanded. The boxes are simple round corner rectangles having a pattern name in the center. Each box represents a specific design pattern instance. The pattern instances are connected to the different elements of a class through unidirectional arrows. PIN kept things simple and focus on the multiple design pattern instances and their roles. The proposed approach is still at infancy stages and work is in progress as mentioned by the author.

Dong et al. [9] presented a new UML profile for the representation and the visualization of the design patterns in their composed form. Few new stereotypes, their corresponding tagged values and constraints are defined in this extended UML notation to explicitly visualize pattern related information in any software design. UML meta-model is used to define such extension in UML profile. This new UML profile represents the role each modeling element (class, attribute, operation) plays in a design pattern. This approach also distinguishes the multiple instances of a design pattern. In addition, authors developed a tool called VisDP [12] for the dynamic visualization of design patterns information. Such information is displayed dynamically when the user moves the cursor on the screen. The applied approach is only limited to the visualization of design patterns information for forward engineering. Furthermore, authors did not focus on the visualization of different types of overlapping in the presented methodology.

Dong [13] proposed another new graphical notation that is an extension to UML. This approach provides a mechanism to visualize each individual pattern in the composition of design patterns by adding the tagged values. These tagged values contain the pattern and/or instance(s) and/or participant name(s) associated with the given class and its operations and attributes. The format of a tag is “pattern [instance]: role” = True/False. For example, if a class is tagged with notation “Adapter [1]: Adaptee” then the class plays a role of Adaptee in the first instance of the Adapter pattern. For the sake of simplicity only the participant name can be shown as it does not create any ambiguity. The author himself determined its limitation as the pattern related information is not as noticeable as the “pattern: role” notation with shading, but he consider it as a tradeoff. This approach provides a mechanism to determine one to one overlapping, but it does not focus on the other two types of overlapping.

Vlissides et al. [3] proposed a notation to explicitly visualize the pattern related information. In this notation, each class is tagged with a gray shaded box containing the pattern related information in the form of “pattern: role”. Each box, associated with a class, contains the pattern name and the role name that this class plays in the associated design pattern. If a class participates in more than one design pattern then all the design patterns in which this class participates will be presented in the same box. For the sake of simplicity, if the class role name is the same as of the design pattern then the design pattern name can be omitted. This notation is more scalable, highly readable and informative. However, the size of the original diagram can significantly be increased as each class has an associated gray box with it. This notation also does not represent the role that an attribute and operation play in a design pattern. There occur multiple instances of a design pattern in a class diagram, but this approach cannot distinguish the multiple instances of a design pattern. Moreover, Dong et al [9] identified the problems related to scanning and reading on the printed media because of gray backgrounds.

Schauer et al. [14] developed a prototype to make the program comprehension simple and efficient by recognizing design patterns and their visualization. For the visualization of design patterns, they proposed pattern enhanced class diagrams that use different colored borders to identify different patterns. This approach provides ease for identifying all the classes participating in a design pattern as this approach is strongly visual. However, this approach cannot identify the role a class, attribute and operation plays in a given design pattern. Moreover, it is really difficult to identify all the design patterns in which a class participates. Authors argue that they enhanced UML representation, but statistically it is difficult to measure improvement in the presented approach. The examples selected for experiments are very small and generalization of approach for large and complex systems is questionable.

Vlissides [3] presented an alternative notation that addresses the limitations of Venn diagram notation. In this notation, dashed ellipses are used to represent the design pattern names. Each ellipse (design pattern) is connected with its participating classes through dashed lines. These dashed lines contain the role names each class plays in a specific design pattern. This approach gives a major breakthrough over Venn diagram notation by specifying the role each class plays in a design pattern. However, the role that attribute or operation plays in a design pattern is not covered in this approach. In addition to this, the scalability issue arises as the system size grows. The design pattern information and the class structure get intermingled and the cluttered lines make it really hard to read and identify the required information [9 13].

Vlissides [3] introduced another intuitive approach to explicitly visualize the design patterns participating in a design diagram in the same report. To distinguish the design patterns from each other, all the classes participating in a design pattern are bounded with different shades of colors. Hence, all the classes participating in a design pattern are easily identified. This approach works well for small number of patterns in the system, but the scalability issue arises when the system size grows. It becomes very difficult to differentiate the overlapping among different design patterns when different classes participate in multiple design patterns. Moreover, this notation does not clearly depict role of different artifacts in the corresponding design patterns. The major focus of authors is to identify the boundary of each design pattern [9, 13].

The concept of composition of design patterns using formal specification is presented by Bayley et al. [4]. Authors applied idea of composition of patterns based on lifting and specialization operations on patterns. The meta-modeling notation GEBNF [16] is used for specification of composition in the design patterns. The applied approach has key focus on composition and formal verification of patterns which can be used for detection of design patterns, but it has no link with the visualization of design patterns during forward and reverse engineering phases which is a major focus of our approach. Composition of patterns is explained using Composite, Strategy and Observer design patterns.

Marie et al. [20] presented a design pattern visualization approach based on different pattern matching views. They used class view, pattern view and Abstract syntax view to represent and visualize information related with design patterns. User can compare candidate patterns with the specification. The experiments are performed on an Observer design pattern using JHotdraw [22] and generalization of composition with other patterns need investigation. The approach did not focus on visualization of operations and different types of overlapping in recognized design patterns. The approach is also limited to visualization for reverse engineering purpose.

III. Comparision of Existing Approaches

UML and non-UML based notations are presented by different researchers for the specification and visualization of information related with design patterns. Non-UML based notations provide option of reasoning, verification and tool support, but they lack support for integration with other tools. These notations also have limitations to specify all features of design patterns. For example, LePUS is not cable to specify all GoF design patterns and their variants [26]. Standard UML based notations are also not capable to model all properties of design patterns, but still they are widely used due to integration of UML tools with other tools. While comparing different notations in this paper, we focus on UML based notations.

An empirical study conducted by Porras et al. [8] concluded that none of the existing notations fits all possible tasks. Therefore, it is important to carefully investigate all the notations. Authors suggested a framework to compare the current and future notations. The findings of study reflect that different notations have their strengths and limitations thus providing a ground to devise new notations that would further overcome identified limitations while combining the best of current notations. We compare the features of existing notations that are presented by different authors [3, 9, 11, 13, 14, 20] on the basis of attributes suggested by the existing framework [8] and by adding new attributes as indicated in Table 1. The major focus in our evaluation is visualizing different types of overlapping in the design of software systems.

Venn diagram-style pattern annotation and Pattern Enhanced Class Diagram are strongly visual approaches, but they do not specify the role each modeling element plays in a design pattern. They are just used to identify the boundary of a design pattern. UML Collaboration Notation is both visual and textual approach representing the role a class plays in a design pattern. However, the role that an attribute or operation plays in a design pattern is not covered in this approach. In addition to this, the scalability issue arises as the system size grows. The design pattern information and the class structure get intermingled and the cluttered lines make its comprehension really hard. Pattern: Role Notation also does not represent the role that an attribute and operation play in a design pattern. This approach is not able to distinguish the multiple instances of a design pattern.

Tagged Pattern Annotation, Tagged Pattern Annotation with shading, Tagged Pattern Annotation with bounding, Tagged Pattern Annotation with new compartments and Stereotype enhanced UML diagrams represent the role each modeling element plays in a design pattern, but the notations make the information really hard to read. Pattern Instance Notation represents the roles each modeling element plays in a design pattern and focus on the multiple design pattern instances, but this approach is still in progress. Furthermore, all these notations do not focus on the visualization of different types of overlapping in design patterns. We present an extended evaluation framework which compare features of all existing notations as given in Table 1.

There are no standard metrics to measure and compare features of different notations such as comprehension, complexity and flexibility. We deeply analyzed features of existing notations and defined our self-scales to measure these features for the purpose of visualization. For example, we defined three scales for measuring the comprehension. These scales are easy, moderate and hard. The easy scale means that comprehension of a notation is user friendly and it is not complex to understand. The moderate scale means that comprehension is between easy and hard scales. The hard scale means that comprehension of a notation is difficult. Similarly, other features are also measured based on our self-scales. The major challenge for visualization approaches is generalization of scalability feature for the visualization of design patterns information. The existing approaches ensure scalability for small and medium size of software packages, but the scalability for large software applications is challenging.

TABLE. I. COMPARISON OF DESIGN PATTERN REPRESENTATION APPROACHES BASED ON DIFFERENT VISUALIZATION ATTRIBUTES

Approaches/ Attributes	**Venn diagram-style pattern annotation [3]**	**UML Collaboration Notation [3]**	**Pattern: Role Notation [3]**	**Pattern Enhanced Class Diagram[14]**	**TPA [13]**	**TPA with shading [13]**	**TPA with bounding [13]**	**TPA with new compartments[13]**	**Stereotype Enhanced UML diagrams [9]**	**Pattern Instance Notation [11]**	**Pattern Matching Views [20]**
Representation Style	Strongly visual	Visual & Textual	Visual & Textual	Strongly visual	Strongly visual	Visual & Textual	Visual & Textual	Visual & Textual	Strongly textual	Visual & Textual	Visual
Visualization Scope	Static	Static	Static	Static	Static	Static	Static	Static	Static & Dynamic	Dynamic	Static & Dynamic
Level of Complexity	Low	Medium	Low	Low	Medium	Medium	Medium	Medium	High	Medium	Medium
Comprehensi-on	Easy	Hard	Easy	Easy	Moderate	Moderate	Moderate	Moderate	Hard	Moderate	Moderate
Tool Support	No	No	No	No	No	No	No	No	Yes	No	Yes
Scalability	Small	Small	Medium	Small	Medium	Small	Small	Medium	Medium	Medium	Small
Flexibility	Less	Less	Medium	Less	Good	Good	Good	Good	Good	Good	Good
Participation	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Composition	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Role	No	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Class Role	No	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Attribute Role	No	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Operation Role	No	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	No
Multiple Instance	No	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	No
1-1 Overlapping	No	No	No	No	Yes	Yes	Yes	Yes	Yes	No	No
1-M Overlapping	No	No	No	No	No	No	No	No	No	No	No
M-M Overlapping	No	No	No	No	No	No	No	No	No	No	No

TPA: Tagged Pattern Annotation

It is visible from features of the state of the art design pattern visualization notations in Table1that there is no notation that can visualize all types of overlapping. The comprehension of different notations varies when the size of an application increases. In order to overcome the limitations of above mentioned design pattern visualization techniques, this paper is intended to propose an approach that integrates the best features of Pattern: Role notation[3] and stereotype enhanced UML diagrams[9] while overcoming their limitations. The proposed approach also appends new functionality regarding visualization of different types of overlapping in UML class diagrams which is important for the comprehension of software applications.

IV. Proposed Hybrid Approach

It is apparent from discussion in Sections 2 and 3 that the current design pattern visualization approaches are unable to capture all the aspects of design pattern visualization which is important for the comprehension of any software application e.g., the role that a class, attribute and operation play in a design pattern. Similarly, there exist multiple instances of a design pattern and different types of overlapping among different classes. The key motivation for this approach is to represent and visualize the pattern related information in the composition of design patterns. Our hybrid notation is elaborated in the following three subsections.

A. *Building on Pattern:Role Notation*

Our proposed notation is given below:

Pattern [Pinstance]: Role [Rinstance]

Where Pattern represents the design pattern name in which a class participates. Pinstance represents the instance of a specific design pattern as there can be multiple instances of a design pattern in the software design. Role represents the role name a class plays in the associated design pattern. Rinstance represents the multiple instances of a class role. We will use this field to visualize different types of overlapping among different classes. A note box containing design pattern information is attached to each class. For the sake of simplicity, if there is only one instance of a design pattern then Pinstance can be omitted. Similarly if there are not multiple instances of a class role then Rinstance can be omitted for ease. For example, file class plays the role of leaf in composite design pattern as shown in Fig. 1. As there is only one instance of leaf therefore Rinstance can be omitted. Also if the design pattern and the class role names are same then the class role can be omitted. For example, directory class plays the role of composite in the Composite design pattern and there is only single instance of the composite design pattern. Thus for the sake of simplicity, the role and the Pinstance fields are omitted as shown in Fig. 1.

The following example further explains how our notation represents information when a single class plays more than one role in different design patterns:

Adapter [1]:Adaptee[1]
Strategy [2]: Context [1]
Bridge [1]:Implementor[1]

Suppose above notational information is attached to a Class A. The notations reflect that Class A plays the role of Adaptee in the first instance of an Adapter design pattern. The same Class A plays the role of Context in the second instance of Strategy design pattern and a role of Implementor in the first instance of Bridge design pattern. The '1' on the right hand side of above notations state that Class A is overlapped in three design patterns with different roles.

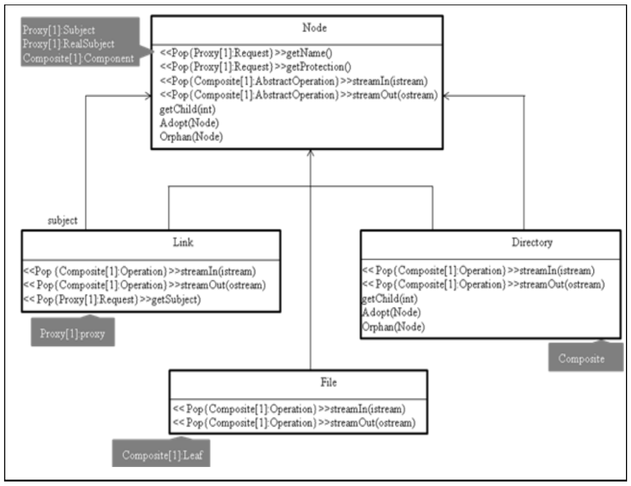

Proxy[1]:Subject
Proxy[1]:RealSubject
Composite[1]:Component
Node
<<Pop(Proxy[1]:Request)>>getName()
<<Pop(Proxy[1]:Request)>>getProtection()
<<Pop(Composite[1]:AbstractOperation)>>streamIn(istream)
<<Pop(Composite[1]:AbstractOperation)>>streamOut(ostream)
getChild(int)
Adopt(Node)
Orphan(Node)
subject
Link
<<Pop(Composite[1]:Operation)>>streamIn(istream)
<<Pop(Composite[1]:Operation)>>streamOut(ostream)
<<Pop(Proxy[1]:Request)>>getSubject)
Proxy[1]:proxy
Directory
<<Pop(Composite[1]:Operation)>>streamIn(istream)
<<Pop(Composite[1]:Operation)>>streamOut(ostream)
getChild(int)
Adopt(Node)
Orphan(Node)
Composite
File
<<Pop(Composite[1]:Operation)>>streamIn(istream)
<<Pop(Composite[1]:Operation)>>streamOut(ostream)
Composite[1]:Leaf

Fig. 1. File System Class Diagram based on our Integrated Approach

B. *Incorporating UML Profile for Design Patterns Visualization*

Stereotypes are used to extend UML profile by defining tagged values and constraints. These tagged values and constraints corresponding to a stereotype get attached to modeling element to which that stereotype is branded [17]. Two stereotypes <<Pat>> for Pattern attribute and <<Pop>> for Pattern operation are defined to explicitly visualize the role that an attribute and operation performs in a design pattern. Each element is associated with its respective stereotype e.g., stereotype <<Pat>> is associated to all such attributes of a class that plays a specific role in a design pattern. Similarly, <<Pop>> stereotype is associated with all operations of a class that are the participants of a design pattern. The tagged values corresponding to these stereotypes are defined in Table 2. The semantics and constraints on applied stereotypes are discussed below:

1) *Semantics*

The detailed semantics of the stereotypes and their corresponding tagged values are given in Table 2. <<Pat>> and <<Pop>> stereotypes are defined to be associated with the attributes and operations of a class that play specific roles in a design pattern respectively. Each stereotype is applied on the corresponding modeling element and the role of that element is identified by the tagged value. The format of the tagged value is Pattern [Pinstance]: Role, Pattern specifies the design pattern name in which this attribute or operation participates, Pinstance specifies the number of design patterns instance to which this attribute or operation belongs to, Role specifies the certain role name that this attribute or operation plays in the design pattern. For example, in Fig. 1, getName () operation of class Node plays the role of Request in Proxy design pattern. There is only one instance of Proxy in the given system. Therefore, the stereotype <<Pop {Proxy [1]: Request}>> is branded to the getName () operation of class Node. It may be possible that an attribute or operation play different roles in different design patterns.

TABLE. II. STEREOTYPES AND TAGGED VALUES ON ATTRIBUTE AND OPERATION

Stereotypes	**Applies to**	**Tagged value**	**Description**
<<Pat>>	Attribute	Pattern[Pinstance]: Role	Identifies that the associated attribute performs the role of Role in this specific instance Pinstance of a design pattern named Pattern
<<Pop>>	Operation	Pattern[Pinstance]: Role	Identifies that the associated operation performs the role of Role in this specific instance Pinstance of a design pattern named Pattern

2) Constraints

We discuss in detail the constraints that are imposed on the stereotypes used by our approach. As stereotypes are associated with modeling elements (attribute and/or operation), constraints also get associated with them. These constraints compel certain kinds of restrictions on the modeling elements. We used Object Constraint Language OCL [18] to write these constraints formally. To define constraints for <<Pat>> and <<Pop>>, we will use standard properties of OCL such as self.a, where a can be a reference or any base class. The constraints for the stereotypes <<Pat>> and <<Pop>> are defined as follows:

<<Pat>>:
self.baseClass = Attribute and self.taggedValue -> exists (tv:taggedValue | tv.name = "Pattern[Pinstance]:Role" and tv.dataValue = Boolean)
<<Pop>>:
self.baseClass = Operation and self.taggedValue -> exists (tv:taggedValue | tv.name ="Pattern[Pinstance]:Role" and tv.dataValue = Boolean)
The Pattern and the Role fields of the tagged values in <<Pat>> and <<Pop>> cannot be empty.
<<Pat>>:
self.taggedValue.name.Pattern -> notEmpty
<<Pop>>:
self.taggedValue.name.Pattern -> notEmpty
<<Pat>>:
self.taggedValue.name.Role -> notEmpty
<<Pop>>:
self.taggedValue.name.Role -> notEmpty
The Pinstance field of the tagged values in <<Pat>> and <<Pop>> can be omitted if there is only one instance of a design pattern. See for example below:
<<Pat>>:
self.taggedValue.name.Pinstance -> isEmpty or self.taggedValue -> exists
(tv:taggedValue | tv.name.instance -> notEmpty)
<<Pop>>:
self.taggedValue.name.Pinstance -> isEmpty or self.taggedValue -> exists
(tv:taggedValue | tv.name.instance -> notEmpty)

The Pinstance field of the tagged values in <<Pat>> and <<Pop>> cannot be omitted if there are multiple instances of a certain design pattern. For example;
<<Pat>>:
self.taggedValue.name -> exists (n1, n2: name | n1.name = n2.name) implies (n1.Pinstance -> notEmpty and n2.Pinstance -> notEmpty and n1.Pinstance != n2.Pinstance)
<<Pop>>:
self.taggedValue.name -> exists (n1, n2: name | n1.name = n2.name) implies (n1.Pinstance -> notEmpty and n2.Pinstance -> notEmpty and n1.Pinstance! = n2.Pinstance)

C. Visualization of Composition

Design patterns are mostly used in a composed form and multiple types of overlapping occur among different instances of design patterns. Visualization of the overlapping in recognized design patterns play an important role for the program comprehension during forward as well as reverse engineering. When the design patterns are composed with each other, there may occur three types of overlapping namely one to one, one to many and many to many overlapping. State of the art design patterns visualization approaches did not pay attention to detect and visualize overlapping. In this paper, our focus is on visualizing all three types of overlapping for forward as well as reverse engineering purposes. We want to clarify that our approach takes extracted result of design pattern recovery tools and then visualize information related with design patterns. One to one overlapping: If there is only one leaf class in the Composite pattern and the composite pattern is composed with the Adapter pattern in such a way that this leaf class is adapted by the adapter then that is a one-to-one overlap. In this case, the same class plays two different roles in two different design patterns. One to many overlapping: If there are multiple leaves of Composite pattern and the Composite and Adapter patterns are composed with each other in such a way that two or more leaves of the Composite pattern are adapted by the same Adapter pattern then this is called one-to-many overlapping. Finally, many to many overlapping: this type of overlapping occurs among patterns when more than one role in a pattern are reused more than one time in another pattern. Zhu et al. [19] presented the composition of Composite and Adapter design patterns with many to many overlapping. In this composition, there are multiple instances of component leaf of Composite design pattern. There are some instances of Leaf class that are adapted by multiple instances of target of Adapter design pattern. Hence, there are multiple targets for multiple leaves. This is an example of many (Targets) to many (Leaves) overlapping.

Figs. 2, 3 and 4 give a view of visualization of one to one, one to many and many to many types of overlapping using our hybrid approach.

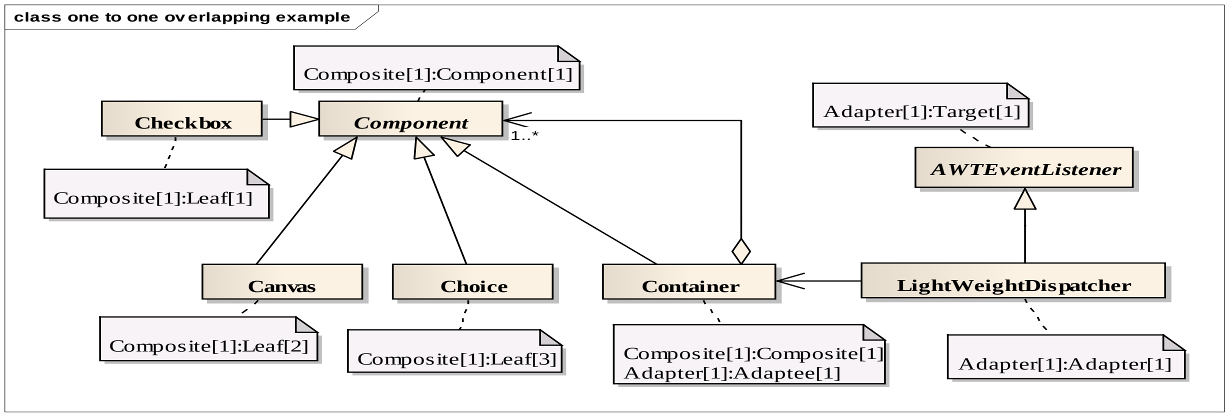


Fig. 2. Visualization of One to One Overlapping in Java.awt

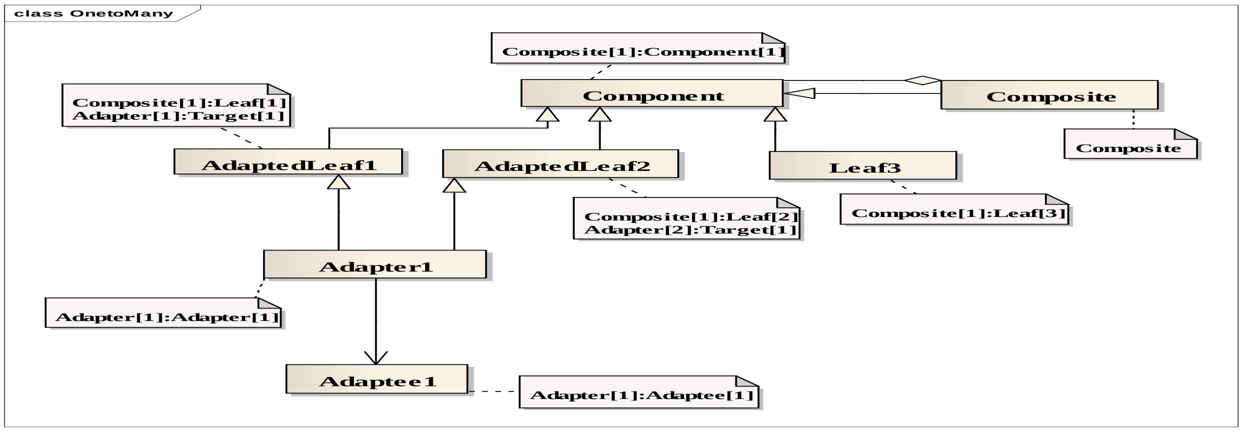


Fig. 3. Visualization of One to Many Overlapping

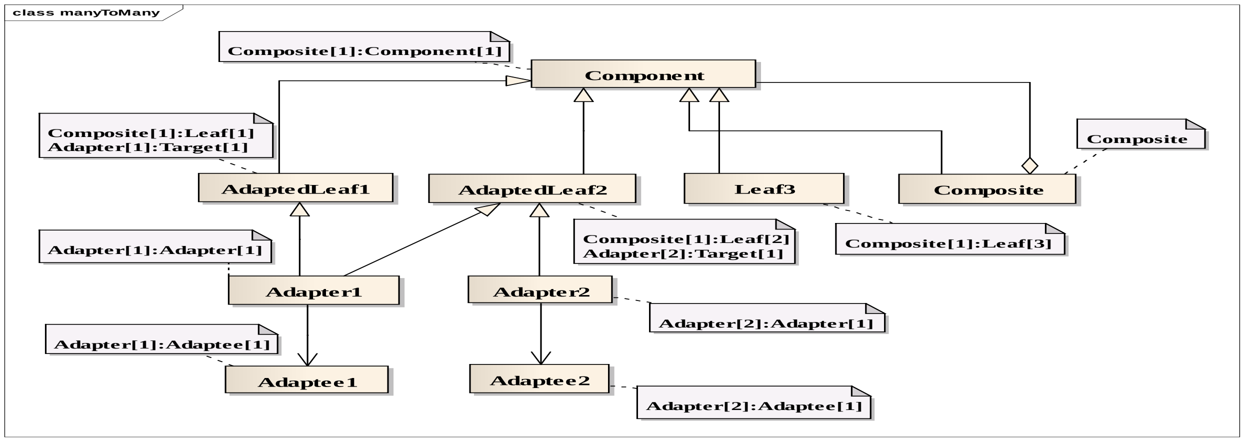


Fig. 4. Visualization of Many to Many Overlapping

V. Prototyping Tool

A prototype tool, VisCDP is developed for the realization of proposed approach. VisCDP is used to visualize design pattern information related to classes, operations and/or attributes in the composition of recognized design patterns. It provides static as well as on demand (dynamic) visualization in UML class diagrams. On demand option is used for filtration and highlighting information about roles participating in different design patterns. For example, by moving the cursor in a class, operation/attribute name, a box with highlighted design pattern information is displayed. These highlighted boxes improve the visibility and comprehension of information. VisCDP supports filtration option on both class and design pattern names and the user can view any specific class and/or design pattern information in a tabular form.

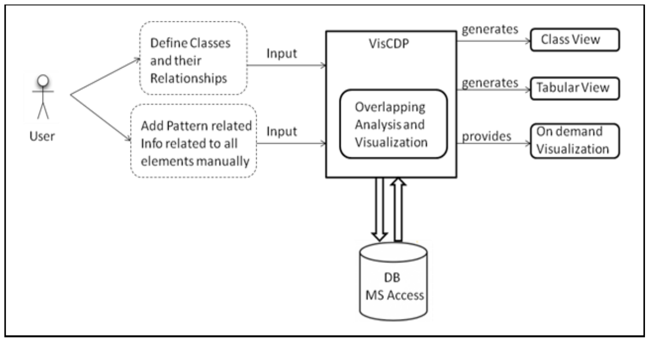
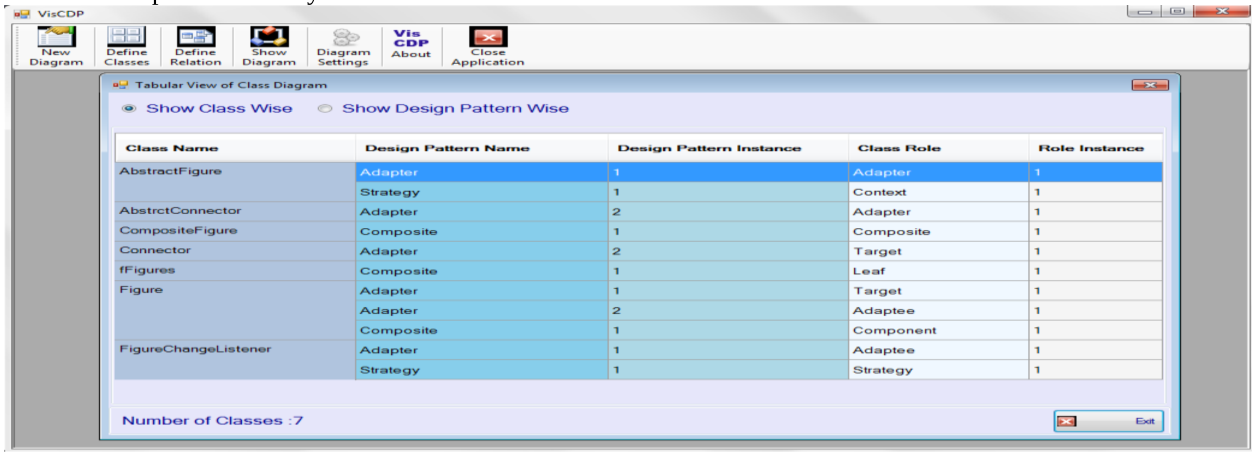

Define Classes and their Relationships
Input
Add Pattern related Info related to all elements manually
Input
User
VisCDP
Overlapping Analysis and Visualization
generates
Class View
generates
Tabular View
provides
On demand Visualization
DB MS Access

Fig. 5. Architecture of VisCDP

The user enters the class and associated design pattern information manually into the tool and then he/she can visualize the output in pictorial and tabular view according tohis/her requirements. Fig. 5 presents the architectural overview of VisCDP. The Visual Studio.NET is used to create the Web forms. The presentation layer for VisCDP is VB.NET forms with user controls. VisCDP takes the input from the user through these forms and then stores this information in a database (MS Access). The user can define the classes, their relationships and all the design pattern information related to classes, operations and attributes. VisCDP displays the original UML diagram and UML diagram with design pattern information related to class and/or operations. It also generates the tabular view of design pattern information related to classes and facilitates on demand visualization.

VisDP is also capable to display design pattern information in a tabular form which is important to know the impact of each class in different design patterns as shown in Fig. 6. The first column in Fig. 6 shows name of a class and the second column show name of design pattern in which a particular class exists. The third column shows the number of design pattern's instance in which a class exists. The last column in Fig. 6 shows that how many roles a particular class is playing in different design patterns.



Class Name	Design Pattern Name	Design Pattern Instance	Class Role	Role Instance
AbstractFigure	Adapter	1	Adapter	1
	Strategy	1	Context	1
AbstrctConnector	Adapter	2	Adapter	1
CompositeFigure	Composite	1	Composite	1
Connector	Adapter	2	Target	1
fFigures	Composite	1	Leaf	1
Figure	Adapter	1	Target	1
	Adapter	2	Adaptee	1
	Composite	1	Component	1
FigureChangeListener	Adapter	1	Adaptee	1
	Strategy	1	Strategy	1

Number of Classes :7

Fig. 6. Design Patterns Information in Tabular Form (Class Wise)

VI. EVALUATION OF APPROACH

The presented approach is evaluated on a JHotDraw-5.1software package which is implemented using different design patterns. This version of JHotDraw-5.1[22] contains 136 classes and total lines of source code are 30860. We partially selected a set of nine classes from this software package to demonstrate our approach as proof of concept. We also implemented other two approaches [3, 9] using same software package. The partial software package design is composed of five design patterns: two instances of Adapter and a single instance of Strategy, Composite and Bridge design patterns. Figure class, playing three roles in three different design patterns, is a central abstraction of the drawing editor framework. It represents a graphical figure that users can work with. The objective of selecting this software package is to evaluate our hybrid approach and to compare the results with the other two approaches [3, 9]. Although, we compare eleven different pattern representation approaches in Table 1, but we selected these two approaches for evaluation and comparison with our approach as these are most representative approaches.

Pattern: Role notation (Gamma's Approach) does not represent the role that an attribute and operation play in a design pattern. Multiple instances of a design pattern may exist in a class diagram, but this approach cannot distinguish the multiple instances of a design pattern. Fig. 7 represents the resulting diagram after implementing the Gamma's approach on our case study.

Stereotype enhanced UML diagrams (Dong's Approach) represent the roles that a class, operation and attribute plays in a design pattern. This approach also distinguishes the multiple instances of a design pattern, but the text overload considerably increases the size of the classes and consequently it becomes really hard to read a design pattern related instances of a design pattern, but the text overload considerably increases the size of the classes and consequently it becomes really hard to read a design pattern related information. in different types of overlapping. Fig. 8 presents the resulting diagram after implementing the Dong's approach on our case study.

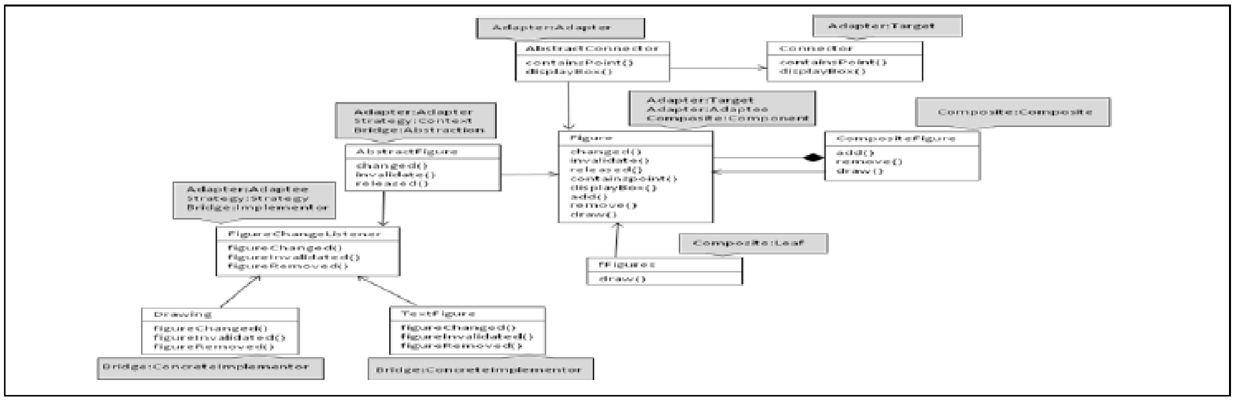


Fig. 7. Gamma's Approach Implemented on JHotDraw-5.1

The proposed approach representing design pattern information on a subset of JHotDraw-5.1 classes is shown in Fig. 9. The notation "Bridge [1]: ConcreteImplementor[1]" attached to class Drawing represents that Drawing class plays the role of ConcreteImplementor in the Bridge. "Bridge [1]:ConcreteImplementor[2]" attached to class TextFigure represents that TextFigure class is the second instance of ConcreteImplementor in the same instance of design pattern Bridge.

Table 3 presents the comparison of Gamma, Dong and our hybrid approach based on the key features used by visualization approaches. One of the major characteristics of our hybrid notation is to represent the multiple instances of a class role that a class plays in different design patterns. This feature exactly determines different types of overlapping i.e. one to one, one to many and many to many which differentiate our approach from state of the art approaches.

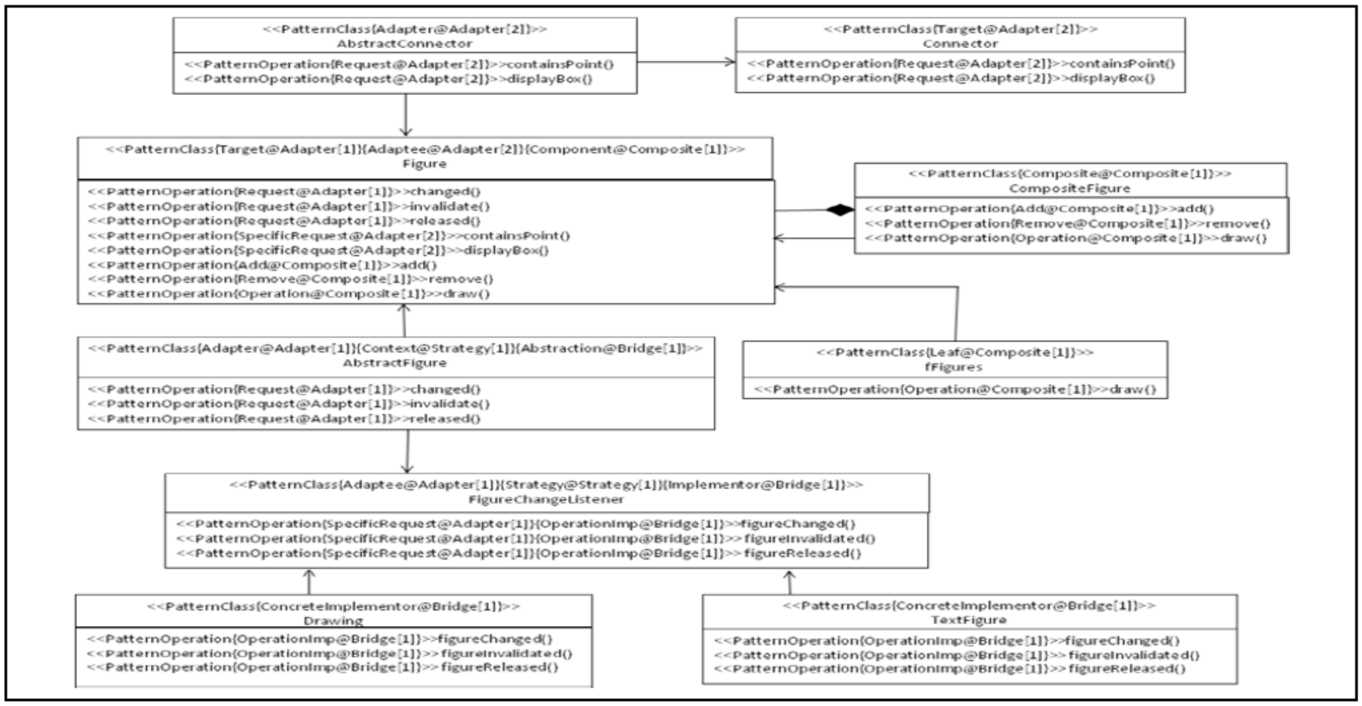

<<PatternClass{Adapter@Adapter[2]}>>
AbstractConnector
<<PatternOperation{Request@Adapter[2]}>>containsPoint()
<<PatternOperation{Request@Adapter[2]}>>displayBox()
<<PatternClass{Target@Adapter[2]}>>
Connector
<<PatternOperation{Request@Adapter[2]}>>containsPoint()
<<PatternOperation{Request@Adapter[2]}>>displayBox()
<<PatternClass{Target@Adapter[1]}{Adaptee@Adapter[2]}{Component@Composite[1]}>>
Figure
<<PatternOperation{Request@Adapter[1]}>>changed()
<<PatternOperation{Request@Adapter[1]}>>invalidate()
<<PatternOperation{Request@Adapter[1]}>>released()
<<PatternOperation{SpecificRequest@Adapter[2]}>>containsPoint()
<<PatternOperation{SpecificRequest@Adapter[2]}>>displayBox()
<<PatternOperation{Add@Composite[1]}>>add()
<<PatternOperation{Remove@Composite[1]}>>remove()
<<PatternOperation{Operation@Composite[1]}>>draw()
<<PatternClass{Composite@Composite[1]}>>
CompositeFigure
<<PatternOperation{Add@Composite[1]}>>add()
<<PatternOperation{Remove@Composite[1]}>>remove()
<<PatternOperation{Operation@Composite[1]}>>draw()
<<PatternClass{Adapter@Adapter[1]}{Context@Strategy[1]}{Abstraction@Bridge[1]}>>
AbstractFigure
<<PatternOperation{Request@Adapter[1]}>>changed()
<<PatternOperation{Request@Adapter[1]}>>invalidate()
<<PatternOperation{Request@Adapter[1]}>>released()
<<PatternClass{Leaf@Composite[1]}>>
fFigures
<<PatternOperation{Operation@Composite[1]}>>draw()
<<PatternClass{Adaptee@Adapter[1]}{Strategy@Strategy[1]}{Implementor@Bridge[1]}>>
FigureChangeListener
<<PatternOperation{SpecificRequest@Adapter[1]}{OperationImp@Bridge[1]}>>figureChanged()
<<PatternOperation{SpecificRequest@Adapter[1]}{OperationImp@Bridge[1]}>> figureInvalidated()
<<PatternOperation{SpecificRequest@Adapter[1]}{OperationImp@Bridge[1]}>> figureReleased()
<<PatternClass{ConcreteImplementor@Bridge[1]}>>
Drawing
<<PatternOperation{OperationImp@Bridge[1]}>>figureChanged()
<<PatternOperation{OperationImp@Bridge[1]}>> figureInvalidated()
<<PatternOperation{OperationImp@Bridge[1]}>> figureReleased()
<<PatternClass{ConcreteImplementor@Bridge[1]}>>
TextFigure
<<PatternOperation{OperationImp@Bridge[1]}>>figureChanged()
<<PatternOperation{OperationImp@Bridge[1]}>> figureInvalidated()
<<PatternOperation{OperationImp@Bridge[1]}>> figureReleased()

Fig. 8. Dong's Approach Implemented on JHotDraw-5.1

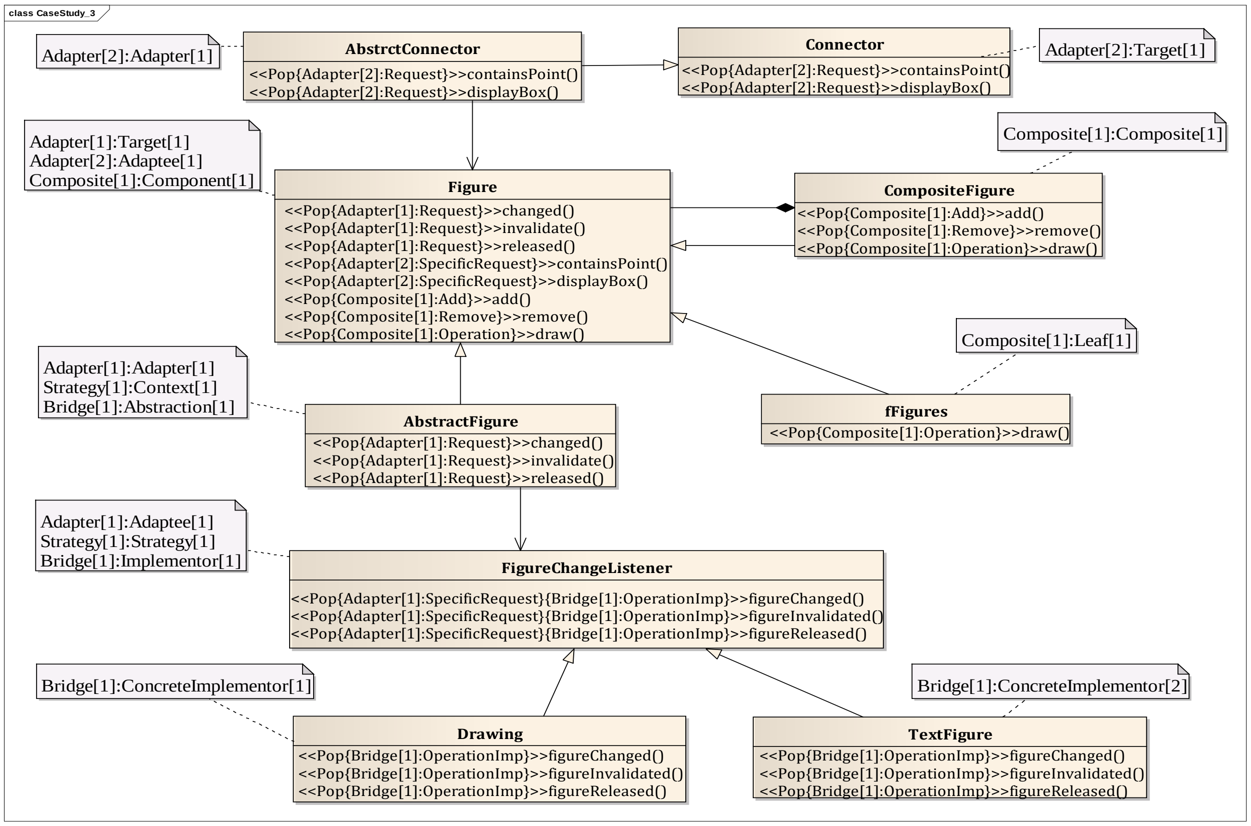

class CaseStudy_3
Adapter[2]:Adapter[1]
AbstrctConnector
<<Pop{Adapter[2]:Request}>>containsPoint()
<<Pop{Adapter[2]:Request}>>displayBox()
Connector
<<Pop{Adapter[2]:Request}>>containsPoint()
<<Pop{Adapter[2]:Request}>>displayBox()
Adapter[2]:Target[1]
Adapter[1]:Target[1]
Adapter[2]:Adaptee[1]
Composite[1]:Component[1]
Figure
<<Pop{Adapter[1]:Request}>>changed()
<<Pop{Adapter[1]:Request}>>invalidate()
<<Pop{Adapter[1]:Request}>>released()
<<Pop{Adapter[2]:SpecificRequest}>>containsPoint()
<<Pop{Adapter[2]:SpecificRequest}>>displayBox()
<<Pop{Composite[1]:Add}>>add()
<<Pop{Composite[1]:Remove}>>remove()
<<Pop{Composite[1]:Operation}>>draw()
Composite[1]:Composite[1]
CompositeFigure
<<Pop{Composite[1]:Add}>>add()
<<Pop{Composite[1]:Remove}>>remove()
<<Pop{Composite[1]:Operation}>>draw()
Composite[1]:Leaf[1]
Adapter[1]:Adapter[1]
Strategy[1]:Context[1]
Bridge[1]:Abstraction[1]
AbstractFigure
<<Pop{Adapter[1]:Request}>>changed()
<<Pop{Adapter[1]:Request}>>invalidate()
<<Pop{Adapter[1]:Request}>>released()
fFigures
<<Pop{Composite[1]:Operation}>>draw()
Adapter[1]:Adaptee[1]
Strategy[1]:Strategy[1]
Bridge[1]:Implementor[1]
FigureChangeListener
<<Pop{Adapter[1]:SpecificRequest}{Bridge[1]:OperationImp}>>figureChanged()
<<Pop{Adapter[1]:SpecificRequest}{Bridge[1]:OperationImp}>>figureInvalidated()
<<Pop{Adapter[1]:SpecificRequest}{Bridge[1]:OperationImp}>>figureReleased()
Bridge[1]:ConcreteImplementor[1]
Bridge[1]:ConcreteImplementor[2]
Drawing
<<Pop{Bridge[1]:OperationImp}>>figureChanged()
<<Pop{Bridge[1]:OperationImp}>>figureInvalidated()
<<Pop{Bridge[1]:OperationImp}>>figureReleased()
TextFigure
<<Pop{Bridge[1]:OperationImp}>>figureChanged()
<<Pop{Bridge[1]:OperationImp}>>figureInvalidated()
<<Pop{Bridge[1]:OperationImp}>>figureReleased()

Fig. 9. 1 Hybrid Approach Implemented on JHotDraw-5

TABLE. III. COMPARISON OF GAMMA, DONG AND OUR HYBRID APPROACH BASED ON KEY FEATURES

DP Representation Features	**Gamma's Approach [3]**	**Dong's Approach[9]**	**Our Hybrid Approach**
Visualization Style	Visual	Strongly Textual	Both Textual and Visual
Overlapping type	No	1-1	All three types
Visualization support	Forward Engineering	Forward Engineering	Forward and Reverse Engineering
Participation	Yes	Yes	Yes
Composition	Yes	Yes	Yes
Class Role	Yes	Yes	Yes
Attribute Role	No	Yes	Yes
Operation Role	No	Yes	Yes
Multiple instances of a design pattern	No	Yes	Yes
Multiple instances of a Class Role	No	No	Yes
Level of Complexity	Low	High	Medium
Comprehension	Easy	Hard	Moderate
Tool Support	No	Yes	Yes

Our proposed approach has visualization support for forward as well as for reverse engineering cycles as compared with approaches of Gamma and Dong. We support all types of overlapping which are important for comprehension of visualization for different instances of design patterns in any software.

Similarly, our approach uses combination of visual and textual aspects of design patterns information for better visualization as compared to previous approaches. Finally, our approach achieved the comprehension at moderate level. We validated comprehension of our approach through a questionnaire. We sent a questionnaire to 20 master students that were studying a course on software visualization at COMSATS Institute of Information Technology. We sent three samples of visualization styles for our approach and other two approaches [3, 9] as a part of the questionnaire. 85% of the students rated comprehension feature of our approach moderate.

Validity is the major concern for researchers and practitioners to validate the results of information retrieval techniques. Regarding construct validity, one of the major threats to the results of our approach is related to design patterns identification from source code and analysis of dependencies as there is a lack of standard definitions for design patterns. The structural and implementation variations are key factors which impact the accuracy of design pattern detection tools. We reduced this threat as we used results of extracted patterns which are already verified. To ensure internal validity, we used JHotDraw-5.1 as a case study. JHotDraw-5.1 is a drawing editor and it is developed by using different design patterns. The source code is available freely for validation of results. However, threats to external validity are related to what extent we can generalize our results. Thus in case of large scale systems, our results for class view may be a threat to the external validity of our visualized results. Regarding reliability validity, we used JHotDraw-5.1 which is open source software and is publically available.

VII. Conclusion and Future Work

The comprehension of large and complex systems based on design patterns is a challenging problem. Different representations of design patterns have been proposed, but each representation has its strengths and limitations. Current design pattern visualization approaches are unable to capture all the aspects of design patterns visualization which is important for the comprehension of any software application e.g., the role that a class, attribute and operation plays in a design pattern. Similarly, there exist multiple instances of a design pattern and different types of overlapping among different classes.

With the critical analysis of state of the art design pattern visualization approaches, we propose an approach that integrates the best features of Pattern: Role notation [3], stereotype enhanced UML diagrams [9] and appends new features to visualize the design patterns in class diagrams. The proposed hybrid notation is used to represent design pattern information related to roles and to visualize different types of overlapping. Stereotypes, their associated tagged values, semantics and constraints are defined to represent the design patterns information related to attributes and/or operations of a class. We used a subset of open source software JHotDraw-5.1 to evaluate our approach and compared the results with the other approaches. The proposed approach improves the visualization of design patterns as compared with previous approaches [3 9]. A prototyping tool named VisCDP is implemented to support our research work and to validate the concept of our hybrid approach. VisCDP is used to visualize design pattern information related to classes, operations and/or attributes in the composition of recognized design patterns. It provides static as well as on demand (dynamic) visualization in class diagrams. It is worthwhile to mention that our current approach is limited only to the visualization of design pattern information in class diagrams and we do not focus on visualization of information in sequence, collaboration and other types of diagrams. We evaluated our approach on a subset of the small scale case study (i.e., JHotDraw-5.1) and scalability of approach for large scale systems is questionable which will be investigated in future. The approach is also limited to visualize the standard representations of GoF patterns and we do not consider variants of same design patterns. In future, we plan to evaluate the scalability of our hybrid approach on large and complex systems.

References

[1] Taibi, T. and D. C. L. Ngo. 2003. Formal Specification of Design Patterns - A Balanced Approach. Journal of Object Technology. (Zurich, Switzerland). 2(4): 127-140.

[2] Dong, J., P. S. Alencar and D. D. Cowan. 2000. Ensuring Structure and Behavior Correctness in Design Composition. In Proc. of 7th IEEE Int. Conf. and Workshop on the Engineering of Computer Based Systems. (Ontario, Canada). pp. 279–287.

[3] Vlissides, J. 1998. Composite Design Patterns (They aren't What You Think). C++ Report. Published by SIGS Publications Group. (NY, USA). 10(4): 45–47.

[4] Bayley, I. and H. Zhu. 2008. On the Composition of Design Patterns. In Proc. of 8th IEEE Int. Conf. on Quality Software. (Washington DC, USA). pp. 27-36. ISBN: 978-0-7695-3312-4.

[5] Hericko, M. and S. Beloglavec. 2005. A Composite Design-Pattern Identification Technique. The Slovene Society Informatica. (Yugoslavia). 29 (4): 469-476. ISSN: 0350-5596

[6] Booch, G., J. Rumbaugh and I. Jacobson. 2005. The Unified Modeling Language User Guide. 2nd Ed. Addison-Wesley. (NY, USA). pp. 104-110. ISBN : 0321267974.

[7] [Dong, J. and K. Zhang. 2003. Design Pattern Compositions in UML. Software Visualization From Theory to Practice. Kluwer Academic Publishers. (Massachusetts, USA). pp. 287–308. ISBN: 1-4020-7448-4.

[8] Porras, G. C. and Y. Gueheneuc. 2010. An Empirical Study on the Efficiency of Different Design Patterns Representations in UML Class Diagrams. Journal of Empirical Software Engineering. (Hingham, USA). 15(5): 493-522.

[9] Dong, J., Y. Sheng and Z. Kang. 2007. Visualizing Design patterns in their Applications and Compositions. IEEE Transactions on Software Engineering. (Los Alamitos, CA, USA). 33(7): 433-453. ISSN: 0098-5589.

[10] Fowler, M. 2002. Patterns of Enterprise Application Architecture. Addison-Wesley. (NY, USA). pp. 45-60. ISBN: 978-0-321-12742-6.

[11] Smith, J. M. 2009. The Pattern Instance Notation: A Simple Hierarchical Visual Notation for the Dynamic Visualization and Comprehension of Software Patterns, In Proceedings of the Workshop Visual Formalisms for Patterns at VL/HCC, pp. 1-12.

[12] Dong, J., Y. Sheng and Z. Kang. 2005. VisDP: A Web Service for Visualizing Design Patterns on Demand. In Proc. of the 6th Int. Conf. on Information Technology: Coding and Computing. (Dallas, Texas, USA). Vol. 2. pp. 385-391.

[13] Dong, J. 2003. Representing the Applications and Compositions of Design Patterns in UML. In Proc. of ACM Symp. on Applied Computing (SAC'03). (NY, USA). pp. 1092-1098. ISBN: 1-58113-624-2.

[14] Schauer, R. and R. K. Keller. 1998. Pattern Visualization for Software Comprehension. In Proc. of 6th IEEE Int. Workshop on Program Comprehension. (IWPC' 98). (Ischia, Italy). pp. 4–12. ISSN: 1092-8138.

[15] Taibi, T. 2006. Formalizing Design Patterns Composition. IEE Proceedings Software. (UK). 153(3): 127–136.

[16] Bayley I., and Zhu. H., Formalising design patterns in predicate logic. In 5th IEEE International Conference on Software Engineering and Formal Methods, 2007.

[17] Berner, S., M. Glinz and S. Joos. 1999. A Classification of Stereotypes for Object-Oriented Modeling Languages. In Proc. of 2nd Int. Conf. on Unified Modeling Language. (Berlin, Germany). pp. 249-264

[18] Warmer, J. B. and A. G. Kleppe. 1998. The Object Constraint Language: Precise Modeling with UML. 1st Ed. Addison-Wesley. (Boston, USA). pp. 60-90. ISBN: 0201379406.

[19] Zhu, H. and I. Bayley. 2012. An Algebra of Design Patterns. ACM Transactions on Software Engineering and Methodology. (NY, USA). Vol. 20. pp. 1-38. ISSN: 1557-7392.

[20] Marie Christin Platenius, Markus von Detten, Dietrich Travkin , Visualization of Pattern Detection Results in Reclipse. Proceedings of the 8th International Fujaba Days, pp. 33-37, May 2011.

[21] Ball, T and Eick, S.G. 1996. Software visualization in the large. IEEE Computer, Vol. 29, issue 4, pp. 33-43.

[22] JHotDraw Home Page: http://www.jhotdraw.org/.

[23] Maplesden, D., Hosking, J.G. and Grundy, J.C., A Visual Language for Design Pattern Modelling and Instantiation, Chapter 2 in Design

Patterns Formalization Techniques, Toufik Taibi (Ed), Idea Group Inc., Hershey, USA, March 2007.

[24] Cacho, N., Sant'Anna, C. Figueiredo E. Garcia A. Batista T. Lucena C., Composing Design Patterns: A Scalability Study of Aspect-Oriented Programming, Proceedings of the 5th international conference on Aspect-oriented software development, pp. 109-121, 2006.

[25] Yacoub, S. M. and H. H. Ammar. 2003. Pattern-Oriented Analysis and Design: Composing Patterns to Design Software Systems. 1st Ed. Addison Wesley Professional. (NY, USA). ISBN: 0201776405.

[26] Amnon, H. Eden, Codecharts: Roadmaps and Blueprints for Object-Oriented Programs. Wiley/Blackwell, 2011.

[27] Byelas, H., Telea, A., 2006. Visualization of Areas of Interest in Software Architecture Diagrams, In SoftVis '06: Proceedings of the 2006 ACM symposium on Software visualization, pp. 105–114.

[28] France, R. B, Kim D-K, Ghosh S, Song E (2004), A UML-based pattern specification technique. IEEE Transaction in Software Engineering, 30(3):193–206.

[29] Kamruddin M. N., Hasan S., Software Visualization Tools for Software Comprehension, The Fourth International Conference on Software, Knowledge, Information Management and Applications, pp. 185-191.

[30] Fontoura, Marcus and de Lucena 2001, Extending UML to improve the representation of design patterns, Journal of Object Oriented Programming, 13(11), pp. 12-19.

[31] Flores, A., Cechich, A., & Aranda, G. 2007, A generic model of object-oriented patterns specified in RSL. Design Patterns Formalization Techniques. IGI Publishing, Hershey, pp. 44-72.

[32] Javed, W. and Elmqvist, N. 2012, Exploring the design space of composite visualization. In Pacific Visualization Symposium (PacificVis), pp. 1-8.

[33] Rasool, G., Umair, M., & Talib, R. 2012, Extended Visualization of Overlapping in Recognized Design Patterns. Journal of Basic and Applied Scientific Research, 2(9), pp. 9080-9087.

[34] Heer, J. and Agrawala, M., 2006, Software design patterns for information visualization, IEEE Transactions on Visualization and Computer Graphics 12(5), pp. 853–860.

[35] Gamma, E., Helm, R., Johnson, R. and Vlissides, J., Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, 1994.