Journal of Object Technology
Published by ETH Zurich, Chair of Software Engineering © Jot, 2010
Online athttp://www.jot.fm

Formal Specification of Structural and Behavioral Aspects of Design Patterns

Shouvik Dey[a] Swapan Bhattacharya[b]

a. IBM India Pvt. Ltd., Kolkata, India
b. National Institute of Technology, Durgapur, India

[Abstract] Design patterns are usually modeled and documented in natural languages and visual languages, such as the Unified Modeling Language. Several UML extensions have been provided to keep track of pattern-related information when a design pattern is applied or composed with other patterns. But several of these are not taken into account while describing a formal specification language. This paper suggests a formal specification language FSDP (Formal Specification of Design Pattern) which combines some of the research works on UML extension mechanisms to formally specify and recognize design pattern from UML class diagram. The FSDP grammar has been verified by ANTLR (ANother Tool for Language Recognition). A simulator tool has been developed from the FSDP grammar to automate the software pattern design techniques. The tool has the capability to create store and retrieve UML class diagrams within design patterns where the model elements of the design patterns play specific roles. The tool verifies the textual contents of the class diagram according to FSDP grammar rules and generates an XML file which contains the detail design. The stored pattern information within the XML file can then be reused by the tool to generate the graphical notation and generate Java codes.

Keywords : Reuse Design Patterns; Pattern composition; UML; Class diagram; Formal representation; ANTLR

1 Introduction

Design patterns [Gamma95] are commonly used in designing large-scale software systems. A pattern is a recurring solution to a standard problem. Since design patterns have been extensively tested and used in many development efforts, reusing them yields better quality software within a reduced time frame. Design patterns are usually modeled and documented in natural languages and visual languages, such as the Unified Modeling Language. UML is a general-purpose language for specifying, constructing, visualizing, and documenting artifacts of software-intensive systems. It provides a collection of visual notations to capture different aspects of the system under development.

Graphical notations include diagrammatic, iconic, and chart-based notations. A graphical notation can be beneficial in many ways. First, it can be used for conveying complex concepts and models, such as object-oriented design. Notations like UML are very good at communicating software designs. Second, it can help people grasp large amount of information more quickly than straight text. Third, as well as being easy to understand, it is normally easier to learn drawing diagrams than writing text because diagrams are more concrete and intuitive than text written in formal or informal languages. Fourth, graphical notations cross language boundary and can be used to communicate with people with different nationalities.

It is seen that the constructs provided by the standard UML and the existing UML extension mechanisms are not enough to visualize design patterns in several applications and compositions. The model elements, such as classes, operations, and attributes, in each design pattern usually play certain roles that are manifested by their names. The application of a design pattern may change the names of its classes, operations, and attributes to the terms in the application domain. Thus, the role information of the pattern is lost. It is not obvious which model elements participate in this pattern. UML does not track pattern-related information when a pattern is applied in a software system or when several patterns are combined. There are several problems when design patterns are implicit in software system designs: first, software developers can only communicate at the class level instead of the pattern level since they do not have pattern-related information in system designs. Second, each pattern often documents some ways for future evolutions, which are buried in system designs. Third, it may require considerable efforts on reverse-engineering design patterns from software system designs [Kelle99]. Hence there is a need to retain the pattern-related information even after the pattern is applied or composed. There has been a lot of research works carried out on this [Fonto03, Dong03, Dong02, Fonto00, France04, Dong07].

As the number of patterns has grown and problems requiring combining patterns surfaced, users started to realize that textual description can be ambiguous and sometimes misleading in understanding and applying patterns. Hence the formal specification of design pattern comes into place. Formal specification of design patterns is not meant to replace the existing textual or graphical descriptions but rather to complement them to achieve well-defined semantics, allow rigorous reasoning about them and facilitate tool support [Taibi03a].

Formal specification of design patterns can enhance the understanding of their semantics. It can be used to help pattern users decide which pattern(s) is (are) more appropriate to solve a given design problem within a context. It can help formalize the combination of design patterns. Finally it can facilitate the development of tools for finding instances of patterns in programs and fine-tuning them to meet pattern specification [Eden01].

A number of formal specification languages have emerged to cope with the inherent shortcomings of textual and graphical descriptions because they focused on specifying the structural and/or behavioral aspect of design patterns but several pattern related information like the role of a participating class or a method in combination of patterns is lost in the existing formal languages. In [Taibi03a] though a balanced approach between the structural and behavioral aspect is provided but the approach does not provide any information like the role(s) of the participating model elements like classes, attributes and the methods in a particular design pattern. The grammar in [Taibi03a] is also not able to provide information on how a class participates in multiple patterns when patterns are combined.

In this paper we propose a language which will bridge the gap between the works of Jing Dong et al. [Dong03,Dong02,Dong07] and T. Taibi et al. [Taibi03a, Taibi03b]. Using this approach we also develop a simulator tool to prove the proposed language.

The rest of the paper is organized as follows: Section 2 represents some of the related works that have been carried out for the extension of UML diagram. Section 3 describes the actual scope of this work. Section 4 gives a brief introduction to FSDP model. Section 5 introduces the proposed FSDP (Formal Specification of Design Pattern) grammar which is written as per ANTLR specification. In section 6 the proposed grammar has been explained elaborately. We have taken a real life example for the case study and illustrate in section 7 using the proposed grammar. Section 8 provides the detail of the simulator tool which we have developed to implement the FSDP grammar and the last section 9 concludes the paper.

2 Related Works

UML extension mechanisms have been used to expand the expressive power of UML to model and represent object-oriented framework [Fonto00], software architecture [Medvi02, Kande00, Zarra01], and agent-oriented systems [Wagne02] when the original UML is not sufficient to represent the semantic meaning of the design. Medvidovic et al. [Medvi02] applied the UML extension mechanism for modeling software architectures. They extended the UML to model software architecture in UML. Kande and Strohmeier [Kande00] extended the UML by incorporating key abstractions in Architecture Description Languages, such as connectors, components and configurations. They focus on how UML can be used for modeling architectural viewpoints. Zarras et al. [Zarra01] applied the UML extension mechanism for architecture description and provided a base UML profile for existing ADLs. Fontoura et al. [Fonto00] proposed a UML extension, called UML-F, to represent object-oriented frameworks. The authors defined a set of new tagged values which can help to represent an object-oriented framework more meaningfully by UML. But the authors failed to give the complete UML profiles for the newly defined stereotypes and tagged values. Wagner [Wagne02] applied the UML extension mechanisms for agent-oriented modeling. A set of new stereotypes are defined to model agent-oriented systems. Jing Dong [Dong03] proposed new stereotypes, tagged-values and constraints to visualize design patterns in composite design patterns. This work uses the UML extension mechanisms to visualize the pattern-related information hidden in a class diagram. They defined new stereotypes, tagged values and constraints, and present a general description of their semantics. It also presents a description of how the UML extensibility mechanisms have been applied in the definition of a UML profile for design patterns. In [Dong02, Dong07] Jing Dong et al. proposed different approaches like Venn diagram style notation etc. to represent design patterns compositions.

In [Gamma95] Rik Eshuis et al. defined a formal execution semantics for UML activity diagrams that is appropriate for workflow modeling. The semantics is aimed at the requirements level by assuming that software state changes do not take time. It is based upon the STATEMATE semantics of state charts, extended with some transactional properties to deal with data manipulation. That semantics also deals with real-time and multiple state instances. They first give an informal description of their semantics and then formalize this in terms of transition systems. They introduced two semantics. The first semantics supports execution of workflow models. Although this semantics is sufficient for executing workflow models, it is not precise enough for the analysis of functional requirements (model checking), since the behavior of the environment is not formalized. They therefore defined a second semantics, which is used for model checking, that extends the first one by formalizing the combined behavior of both the system that the activity diagram models and the system's environment. Their semantics is different from the OMG activity diagram semantics [OMG09], because they map activities into states, whereas the OMG maps them into transitions. The OMG semantics implies that activities are done by the WFS (Work Flow System) itself, and not by the environment. In their semantics, activities are done by the environment (i.e. actors), not by the WFS itself. T. Taibi and D.C.L. Ngo [Taibi03a,Taibi03b] proposed a simple yet Balanced Pattern Specification Language (BPSL) that is aimed to achieve equilibrium by specifying structural and behavioral aspects of design patterns. BPSL combines two subsets of logic, one from First Order Logic (FOL) and one from Temporal Logic of Actions (TLA). France et al. [France04] presented a rigorous and practical technique for specifying pattern solutions expressed in the UML. The specification technique paves the way for the development of tools that support rigorous application of design patterns to UML design models. The technique has been used to create specifications of solutions for several popular design patterns. They illustrated the use of the technique by specifying observer and visitor pattern solutions. In [Bayle07] I. Bayley et al. deployed predicate logic to specify conditions on the class diagrams that describe design patterns.

Eden et al. proposed LanguagE for Patterns' Uniform Specification (LePUS) [Hirsh99] which is a symbolic logic language for the specification of recurring motifs in object oriented architectures. LePUS' constructs express concisely fundamental elements of O-O architecture, such as inheritance-class-hierarchies, and correlations, such as isomorphisms between methods, classes, and hierarchies. LePUS derives from Higher-Order logic and focuses only on specifying the structural aspects of design patterns. Distributed Co-operation (DisCo) [Mikko98] was derived from TLA (Temporal Logic of Actions) and was designed to specify reactive systems, which are in constant interaction with their environment and therefore have a predominantly behavioral aspect. DisCo has little (almost no) support for specifying the structural aspect.

Design patterns document good solutions to recurring problems in a particular context. Composing design patterns may achieve higher level of reuse by solving a set of problems. Design patterns and their compositions are usually modeled by UML diagrams. When a design pattern is applied or composed with other patterns, the pattern-related information may be lost because traditional UML diagrams do not track this information. Thus, it is hard for a designer to identify a design pattern when it is applied or composed. In [Dong02] Jing Dong presented notations to explicitly represent each pattern in the applications and compositions of design patterns. The notations allow maintaining pattern-related information. Thus, a design pattern is identifiable and traceable from its application and composition with others.

3 Scope of Work

Our work is mostly inspired by the work of T. Taibi et al. [Taibi03a,Taibi03b] and Jing Dong et al. [Dong03, Dong07]. Jing Dong et al. propose new stereotypes, tagged values and constraints, and present a general description of their semantics. They provide a description of the tagged pattern notation in terms of a UML profile. These tagged values define exactly what role a class, an attribute or an operation plays in a design pattern. But their work only describes behavioral representation of design patterns, neither has it specified structural aspects like how different classes are interrelated etc. nor they provide any formal language to describe this. On the other hand T.Taibi et al. propose a formal specification language BPSL for design patterns that is aimed to achieve equilibrium by specifying both structural and behavioral aspects of design patterns. BPSL combines two subsets of logic, one from First Order Logic (FOL) and one from Temporal Logic of Actions (TLA). But BPSL does not take into account the behavioral role of model elements such as Class, Interface, Attribute or Operations in design pattern. BPSL does not provide information on how a model element plays different roles when several patterns are composed which is specified in Jing Dong's work [Dong03, Dong07]. Moreover BPSL is not able to represent composition of design patterns and it cannot handle the case when there are multiple instances of the same pattern in the same design. In this work we have tried to bridge that gap among these works by proposing a new formal specification language which will combine most of the works of [Dong03,Taibi03a,Taibi03b,Dong07] as well as the missing works mentioned above within a single specification language. Hence FSDP will help to represent design pattern in a more informative and formal way comparing to other existing specification languages when several patterns are composed and combined in a design.

The main goal of this work is to provide a technique for modeling and designing of systems. The primary objective is to be able to represent both the structural and behavioral aspects of design patters in a formal way using the UML extension mechanism. As no other existing language incorporates the work on extension mechanism provided by Jing Dong et al. and T. Taibi et al. we have proposed a language which will help pattern users to recognize and represent pattern related information more clearly when patterns are composed.

To prove and verify the grammar we have developed a simulator GUI (Graphical user Interface) tool which is able to take design pattern class diagrams as input and verify if the text entered within the class diagram are according to the language specified by the proposed grammar. If the textual notations are parsed successfully then we can store this pictorial representation of design pattern combination within an XML file for reuse. Pattern users can anytime retrieve the previously stored system design from that XML file. The simulator is also capable of generating Java code from the design pattern stored hence minimize the work for developer.

4 Introduction to FSDP

We know that Design Patterns are usually modeled and documented in natural languages and visual languages. Hence our model will expect natural language and visual language as input from the user. We restrict our model to take English language as natural language and UML Class diagram as the visual language.

Graphical notations are used mainly for proper and clear description of several design patterns. The graphical notations help visualize the system design. Graphical notations such as UML Class diagrams, Sequence diagrams etc. are generally used. FSDP will use the textual content of the UML class diagrams and represent it in a formal way. We will represent the structural aspects like the classes, methods, attributes in a formal way as well as the behavioral nature like the relationships, association, and cardinality among the participating classes. A simulator GUI has been developed which is capable of implementing the proposed mechanism and store the design pattern in the system. The GUI is also able to generate Java code from the stored design pattern. Hence the reusability can be achieved.

5 Formal Specification of Design Pattern

The grammar to verify the token flow mechanism of FSDP is provided below. The grammar is verified by ANTLR (**ANother Tool for Language Recognition**) which is a recursive-descent powerful parser and translator generator tool, akin to the venerable lex/yacc duo, that lets one define language grammars in either ANTLR (http://www.antlr.org/) syntax (which is YACC and EBNF(Extended Backus-Naur Form) like) or a special AST(Abstract Syntax Tree) syntax. ANTLR implements a PRED-LL(k) parsing strategy and affords arbitrary look ahead for disambiguating the ambiguous.

A compiler has two parts lexer and parser. The job of the lexer is to quantify the input stream of characters into discrete groups called tokens. A lexer usually generates errors pertaining to sequences of characters it cannot match to a specific token type defined by one of its rules. Languages are described by a grammar and the grammar determines exactly what defines a particular token and what sequences of tokens are decreed as valid. The parser organizes the tokens it receives into the allowed sequences defined by the grammar of the language. If the language is being used exactly as is defined in the grammar, the parser will be able to recognize the patterns that make up certain structures and group these together. If the parser encounters a sequence of tokens that match none of the allowed sequences of tokens, it will issue an error and perhaps try to recover from the error by making a few assumptions about what the error was.

Here we have proposed the FSDP specification of the lexer as well as the parser which is then verified by the ANTLR tool. The character set of the proposed grammar includes the set {A-Z, a-z, 0-9} along with some special characters {**. , ; : { } ( ) | _/**}. The terminals and string literals are in capital and bold, non terminals in small.

The FSDP grammar of the Parser and the Lexer is written in a file named as FSDP.g under C:\Shouvik\FSDP\ directory.

The Parser and Lexer part of the grammar are shown in Table 1 and Table 2 respectively.

Table 1: FSDP specification of the Parser

```
    class FSDPParser extends Parser;
    options { k=2;}
    validPattern : structure behavior;
    structure : validClass;
    validClass : classDecl (instance)? (validMethod)?;
    classDecl : className patternDecl (COLON ("class" | "interface"))?;
    patternDecl : LEFTBRACE patternName (patternInstance)? SLASH role (COMMA
patternName
     (patternInstance)? SLASH role)*       RIGHTBRACE ;
    patternInstance : LEFTBRACKET instanceNo RIGHTBRACKET;
    instance : (instanceType COLON instanceName (patternDecl)?)+;
    validMethod : (methodName parameterDecl (patternDecl)? (COLON returnType)?)+;
    parameterDecl : LEFTPAREN (parameterType parameterName (COMMA parameterType
     parameterName)*)? RIGHTPAREN;
    behavior : ((dependency)?) => (dependency)? |((inheritance)?) => (inheritance)? |
((implementation)?) =>
     (implementation)? | (association)?;
    dependency : className className;
    inheritance : className className;
    association : className relationship className;
    relationship : "ONE-TO-MANY" | "MANY-TO-ONE" | "MANY-TO-MANY";
    implementation : className interfaceName;
    instanceType : STRING;
    returnType : STRING;
    instanceName : STRING;
    interfaceName : STRING;
    attribute : STRING;
    className : STRING;
    methodName : STRING;
```

```
methodRole : STRING;
role : STRING;
parameterType : STRING;
parameterName : STRING;
patternName : STRING;
instanceNo : INTEGER;
```

Table 2: FSDP specification of the Lexer

```
class FSDPLexer extends Lexer;
 options { k=2;}
WS : ( ' ' | '\t' | '\f' | ( "\r\n" | '\r' | '\n') { newline(); } )
     { $setType(Token.SKIP); }  ;
 STRING : (CHAR)+;
 LEFTBRACE : '{';
 RIGHTBRACE : '}';
 LEFTPAREN : '(';
 RIGHTPAREN : ')';
 LEFTBRACKET : '[';
 RIGHTBRACKET : ']';
 COMMA : ',';
 COLON : ':';
 SEMICOLON : ';';
 INTEGER : (DIGIT)+;
 SLASH : '/';
 STRING : ('a'..'z'|'A'..'Z') ('a'..'z'|'A'..'Z' | '|' | '_' | '.' | '-' | '0'..'9')*;
 protected
 DIGIT : ('0'..'9');
```

To prove this is a valid grammar we have verified this with the ANTLR tool. We have downloaded antlr.jar version 2.7.7 from `http://www.antlr.org/` and add this jar in the CLASSPATH variable. This jar includes all the required files to verify if a grammar written in ANTLR specific language compiles successfully. To verify the grammar and generate the Parser and Lexer we have executed the following command from the command prompt :

```
C:\Shouvik\FSDP> java antlr.Tool FSDP.g
```

and the result is as below:

```
ANTLR Parser Generator    Version 2.7.7 (20060906)    1989-2005
```

As no error generates, this signifies the grammar we have proposed is successfully compiled and satisfied by the ANTLR tool. The syntax as well as the semantics of the proposed grammar are ok and are according to the ANTLR specification. This command generates 6 files in the background. They are FSDPLexer.java, FSDPLexer.smap, FSDPParser.java, FSDPParser.smap, FSDPParserTokenTypes.java and FSDPParserTokenTypes.txt. As we are interested in building the simulator in Java coding standard we issued the

```
java antlr.Tool FSDP.g
```

command to generate the Lexer and Parser files in Java only.

The FSDPParserTokenTypes.java and FSDPParserTokenTypes.txt files contain the information of the terminals and string literals. For example the content of FSDPParserTokenTypes.txt is shown in Table 3.

Table 3: Content of FSDPParserTokenTypes.txt file

```
// $ANTLR 2.7.7 (20060906): FSDP.g -> FSDPParserTokenTypes.txt$
FSDPParser    // output token vocab name
COLON=4
LITERAL_class="class"=5
LITERAL_interface="interface"=6
LEFTBRACE=7
SLASH=8
COMMA=9
RIGHTBRACE=10
LEFTBRACKET=11
RIGHTBRACKET=12
LEFTPAREN=13
RIGHTPAREN=14
"ONE-TO-MANY"=15
"MANY-TO-ONE"=16
"MANY-TO-MANY"=17
STRING=18
INTEGER=19
WS=20
SEMICOLON=21
DIGIT=22
```

FSDPParser.java and FSDPLexer.java files contain all the methods for lexical analyzing and parsing the input text stream respectively. For example the FSDPParser.java contains the method patternInstance() for parsing the patternInstance rule from the incoming tokens. The content of that method is shown in Table 4. Similarly for each and every production rule there exists a corresponding java method in FSDPParser.java file.

Table 4: patternInstance method of FSDPParser.java

```
public final void patternInstance() throws RecognitionException, TokenStreamException {
        try {      // for error handling
                    match(LEFTBRACKET);
                    instanceNo();
                    match(RIGHTBRACKET);
        }
        catch (RecognitionException ex) {
                    if (inputState.guessing==0) {
                                reportError(ex);
                                recover(ex,_tokenSet_7);
                    } else {
                      throw ex;
                    }
        }
    }
```

6 Illustration of the Language

Till now we were discussing the grammar and how to generate the parser and lexer to accept the language. Now let us illustrate the language and describe how it really helps us to achieve the goal. There are two stages involved in the specification of FSDP language. Formation of the tokens is done by the lexer and then parser checks if the tokens conform to the syntax of the language defined by the grammar. Let's take a look at the Parser. ANTLR has some inbuilt classes; Parser class is one of them. To create a user defined parser the new parser class has to extend from ANTLR Parser class. Hence our parser generator class FSDPParser extends from Parser class so that it inherits all the required features of its parent class. ANTLR affords arbitrary look ahead for disambiguating the ambiguous. Options section is used to declare how many characters parser should look ahead to make a decision. Our grammar is bold enough to take decision and disambiguate by looking only next two characters. The tokens section explicitly defines literals. The root of the parser rule starts with validPattern. The rule says that a design pattern should have the structural as well as the behavioral information.

validPattern : structure behavior;

The structure part of the rule holds all the necessary information about the participating classes, their attributes and methods while the behavior rule provides the semantic of how the participants cooperate to carry out their responsibilities. Let's discuss in detail the structure as well as the behavioral semantics.

structure : classDecl (instance)? (validMethod)?;

To define a structure of a design pattern there should be at least a valid class declaration. There may or may not be instance variables or even methods declared inside the class but a valid class must be declared to represent a design pattern. The term (instance)? or (validMethod)? signifies that they are optional. If they are present in the input token stream parser will also accept those tokens and if they are not present parser will look for next token to be matched with the next rule.

The class is declared in the following production rule.

classDecl : className patternDecl (COLON ("class" | "interface"))?;

which signifies that a class declaration consists of a class name and the declaration of the design pattern(s) in which the class performs certain role(s). At the end there is an option to mention whether the declaration is meant for a class or an interface by specifying a COLON and mentioning either "class" or "interface" literal. If nothing is mentioned at the end we will assume it as the declaration of a class.

The patternDecl production rule is:

patternDecl : LEFTBRACE patternName (patternInstance)? "/" role (COMMA patternName (patternInstance)? "/" role)* RIGHTBRACE ;

and the patternInstance production rule is declared as follows:

patternInstance : LEFTBRACKET instanceNo RIGHTBRACKET;

The patternInstance non terminal rule is made optional. This is because if a class participates in only one instance of the design pattern then there is no need of explicitly mentioning the pattern instance. If there is no such entry we will assume that only one instance of the design pattern is available in the class design.

Now let us clarify the above mentioned three production rules by giving a small example. The combination of these three rules defines how participating class level information can be achieved from composition of design patterns. For example let us assume a class XYZ participates in the first instance of Singleton design pattern where it performs the role of a UniqueInstance and the same class XYZ also plays the role of a ConcreteFactory in the first instance of Abstract Factory design pattern. Now this information can be formally written as per the three production rules:

classDecl : XYZ { Singleton[1]/UniqueInstance , Abstract Factory[1]/ConcreteFactory } : class

Let us now come to the other part of the structure rule. The production rule “instance” is defined as:

instance : (instanceType COLON instanceName (patternDecl)?)+;

Let us take an example for the declaration of an instance of class XYZ. If singletonData is an instance of integer type defined in XYZ which plays the role of UniqueInstance in the first instance of the Singleton design pattern the instance rule will expect the token stream as below:

instance : integer : singletonData { Singleton[1] / UniqueInstance }

Note that the pattern declaration related information patternDecl for instance type is marked as optional because if the class containing the instance participates in only one design pattern that will not carry any extra information to pattern users.

The last part of the structural production rule is to define methods of a class which is given below.

validMethod : (methodName parameterDecl (patternDecl)? (COLON returnType)?)+;

A valid method consists of a method name, the parameter declaration, the declaration of the design pattern and the return type of the method. Like instance declaration the valid method may or may not include the pattern related information if the class participates in a single design pattern. The method also may declare the return type. If nothing is specified we assume that a void type data is returned from the method call. The "+" symbol signifies that the more than one methods can be declared simultaneously under the same class and the FSDP grammar will acknowledge multiple method declarations.

The parameter declaration production rule of valid method is like the following:

parameterDecl : LEFTPAREN (parameterType parameterName (COMMA parameterType parameterName)*)? RIGHTPAREN;

which means that a valid method may have one or more parameters or have no parameter at all.

Now we have completed the discussion of the formal approach of representing the structural part of pattern related designs. We have mostly followed Jing Dong's [Dong03] UML extension mechanism and proposed the formal way of representing the same work. Now we are going to discuss elaborately the formal approach of the behavioral part of the design pattern.

As already mentioned the behavior rule is defined as:

behavior : ((dependency)?) => (dependency)? | ((inheritance)?) => (inheritance)? | ((implementation)?) => (implementation)? | (association)?;

The rule denotes a syntactic predicate (aka "guess" mode) which basically says first try to match dependency part, if it works, use it, otherwise, try the next alternative which is the inheritance part. Then search for tokens which match with implementation rule and finally search for last set of tokens which match the association production rule. The main advantage of using the syntactic predicate in ANTLR is that compiler can backtrack if the matching is not successful and try for the next matching.

The behavioral aspect of the rule consists of the information on how the participating classes are interrelated with each other in the pattern. One class may be dependent on another class that is for example if an instance of class B is used in some methods of class A of other design pattern that means there is a dependency relationship exists between class A and class B where A is dependent on B. One class may inherit the characteristics from some other class. There may be situations where one class is associated with more than one instances of other class in the same design pattern. All the above behavioral interactions can be achieved by our proposed grammar. Normally when several design patterns are composed these information get lost and UML 2.0 does not have the mechanism to preserve these pattern related information. But if the design can be defined using the grammar and the user interface then all these information can be preserved and retrieved for future reuse.

The behavioral rule first looks for dependency information in the pattern. The dependency information is provided by

dependency : className className;

which denotes that if X and Y are two classes and the compiler gets input tokens as

dependency : X Y

this signifies class X is dependent on class Y. Similarly the inheritance rule looks like

inheritance : className className;

i.e. the first class inherits the features from the second class.The implementation rule provides information of a class which wants to implement an interface.

implementation : className interfaceName;

The implementation production rule suggests that the class className implements the interface interfaceName. The association information between two classes can be achieved by the following association rule.

association : className relationship className;

where the relationship production rule is defined as:

relationship : "ONE-TO-MANY" | "MANY-TO-ONE" | "MANY-TO-MANY";

This signifies that the first class is associated with the second class by either ""ONE-TO-MANY" or "MANY-TO-ONE" or "MANY-TO-MANY" cardinality that means the number of instances of the first class is associated with how many instances of the second class. If only one instance of class XYZ is associated with multiple instances of class ABC then the FSDP representation will be:

association : XYZ ONE-TO-MANY ABC

7 Case Study

In this section we will take a case study and explain in detail how the design pattern can be represented formally using the FSDP grammar. Let us assume a system design that manages the connections to different types of databases, such as Oracle and DB2 in Fig 1. This system provides a connection pool for accessing each type of database. The connection pool restricts a limit number of accesses to a database and reuses connections to the database. The system has the capability to handle different types of database connections. The ConnectionPool class defines an interface for the creation of a connection pool for the appropriate type of database. The concrete classes, OracleConnectionPool and DB2ConnectionPool, use the createConnection operation to create the corresponding connections, OracleConnection and DB2Connection, respectively. All connection instances have the same interface which is defined in the Connection class.

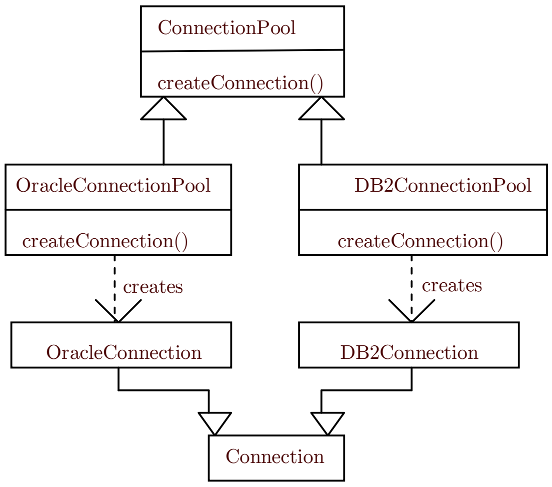

Fig. 1 Connection Pool for Database

Fig. 1 is not expressive enough to provide answer to the missing pattern related information like the role of participating elements such as class, methods in the design pattern. Moreover this is the scenario of combination of design patterns where more than one design patterns are composed and some of the participating classes represent more than one design pattern simultaneously and hence each of these classes have multiple roles, each of the roles corresponding to one pattern. But this information is not reaching to the pattern users. Two design patterns, Abstract Factory and Singleton are applied in the system design.

The ConnectionPool, OracleConnectionPool and DB2ConnectionPool classes play the roles of abstract and concrete factories, whereas the Connection, OracleConnection and DB2Connection classes play the roles of abstract and concrete products in the Abstract Factory pattern, respectively. OracleConnectionPool and DB2ConnectionPool are the Singleton classes, which restrict only a limited number of connections for each database. Hence OracleConnectionPool and DB2ConnectionPool also represent Singleton design pattern. Apart from the pattern related information the role of each of the participating class and its methods are missing in the diagram. Fig 2 gives the solution of this issue and the textual representation in the diagram follows FSDP specification.

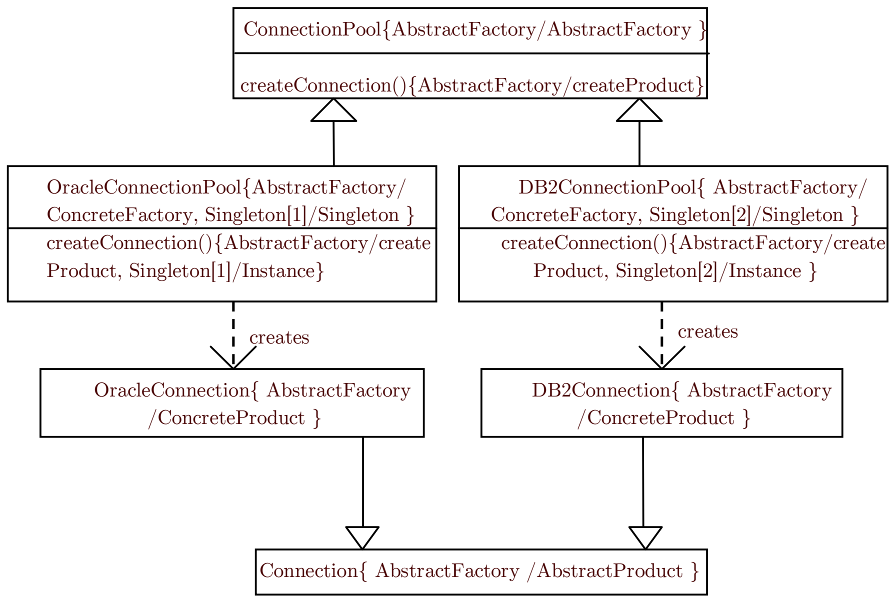
ConnectionPool{AbstractFactory/AbstractFactory }
createConnection(){AbstractFactory/createProduct}
OracleConnectionPool{AbstractFactory/ ConcreteFactory, Singleton[1]/Singleton }
createConnection(){AbstractFactory/create Product, Singleton[1]/Instance}
DB2ConnectionPool{ AbstractFactory/ ConcreteFactory, Singleton[2]/Singleton }
createConnection(){AbstractFactory/create Product, Singleton[2]/Instance }
creates
creates
OracleConnection{ AbstractFactory /ConcreteProduct }
DB2Connection{ AbstractFactory /ConcreteProduct }
Connection{ AbstractFactory /AbstractProduct }

Fig. 2 Connection Pool with proposed notation

Let us take class OracleConnectionPool. It participates in two design patterns AbstractFactory and Singleton. Using the proposed specification the class is now expressed as OracleConnectionPool{ConcreteFactory/AbstractFactory, Singleton[1]/Singleton } which signifies that this class is playing dual roles one as a ConcreteFactory under design pattern AbstractFactory and the other role is a Singleton under the first instance of Singleton design pattern.

Hence using this specification composition of design patterns can be represented efficiently. Also information regarding pattern instances can be represented following the FSDP specification. The createConnection() method of OracleConnectionPool class also plays dual role in the composition of two design patterns. The dual role createProduct and Instance are represented as createConnection() { AbstractFactory/ createProduct, Singleton[1]/Instance }.

Hence the pattern related information is not lost while patterns are composed and combined.

Till now we have shown how representation of UML class diagram can be extended to visualize design pattern related information in pattern compositions. In this representation we have used the concept of Jing Dong's work [Dong03,Dong07] and modified as per our need. The graphical representation in Fig. 2 shows both the structural and behavioral aspects of design patterns. Now we use FSDP specification to represent this graphical notation and verify it using the ANTLR tool. We mainly concentrate on the structural and behavioral representation of the classes participates in the system design pattern in Fig. 2. The structure consists of classDecl, instance and validMethod declaration. Here some of the classes in Fig. 2 have methods declared but no attributes are declared in any of the classes.

Table 5 shows the FSDP representation of the system.

Table 5: FSDP specification of the system

```
        classDecl  : ConnectionPool{AbstractFactory / AbstractFactory }
        validMethod :  createConnection(){AbstractFactory / createProduct }

        classDecl:   OracleConnectionPool  {  AbstractFactory   /  ConcreteFactory,  Singleton[1]  /
Singleton }
        validMethod : createConnection() {AbstractFactory / createProduct, Singleton[1] / Instance}

        classDecl  :    DB2ConnectionPool  {  AbstractFactory  /  ConcreteFactory,  Singleton[2]  /
Singleton }
         validMethod  :  createConnection(){AbstractFactory  /    createProduct,    Singleton[2]  /
Instance }

        classDecl: OracleConnection {AbstractFactory / ConcreteProduct}
        classDecl: DB2Connection { AbstractFactory / ConcreteProduct}
        classDecl:  Connection { AbstractFactory / AbstractProduct }

        dependency : OracleConnectionPool  OracleConnection;
        dependency : DB2ConnectionPool  DB2Connection;
        inheritance : OracleConnectionPool  ConnectionPool
        inheritance : DB2ConnectionPool  ConnectionPool
        inheritance : OracleConnection  Connection
        inheritance : DB2Connection  Connection
```

Table 5 reveals that ConnectionPool class participates in a single design pattern AbstractFactory and plays the role of AbstractFactory under the pattern and consists of only one method named as createConnection and role or responsibility of which is createProduct. Similarly we can say that OracleConnectionPool class participates in two design patterns AbstractFactory and Singleton and hence this class plays dual role, ConcreteFactory role under AbstractFactory pattern and Singleton role under the first instance of Singleton design pattern. Also the method createConnection() in this class performs dual role : createProduct in AbstractFactory pattern whereas Instance role under the first instance of Singleton pattern.

The behavioral aspect of the design shows how the classes DB2Connection, OracleConnection, DB2ConnectionPool, Connection, OracleConnectionPool and ConnectionPool are interrelated with each other. It is clear from Table 5 that OracleConnectionPool is dependent on OracleConnection similarly DB2ConnectionPool is dependent on DB2Connection. The system uses inheritance where OracleConnectionPool and DB2ConnectionPool inherit from ConnectionPool class whereas OracleConnection and DB2Connection inherit from Connection class.

As there is no information provided in Fig 1 regarding the association relationships among the classes there will be no FSDP representation for association rule.

Now that we have specified the system in FSDP language. We need to verify whether the system specification is really accepted by the FSDP parser or not and how the parser works to identify an input string. This is discussed in the next section.

8 Implementation of FSDP

Till now we have discussed how we can represent design pattern formally using FSDP grammar. We have developed a simulator GUI tool that is able to provide a designing environment where pattern developers can represent the system design patterns using the UML class diagram and the tool will be able to verify the textual content whether it is formally represented as per FSDP grammar specification or not. The interface supports users to draw the graphical notations of classes, methods they contain, relationships as well as the association among the classes. The textual information entered in the class diagram of the design pattern is verified by the ANTLR parser and then stored in an XML file in the system. The XML file can be reused further to generate the graphical representation in the GUI tool. The tool has also the feature of generating java code which reflects the structural as well as behavioral aspects of the system design. We have chosen XML to store the pattern information as because XML is a standard language which was designed to transport and store data. XML data is stored in text format. This makes it easier to expand or upgrade to new operating systems, new applications, or new browsers, without losing pattern information. Hence the design created and stored using the proposed grammar using any tool, written in any language (for example, we have created a Java based tool) can be extracted in any other system using any of the existing XML parsers and it can then be reused for reverse engineering.

In brief, the simulator tool developed by us serves multiple purposes as following:

- Creates a visual representation of system design pattern using UML class diagram and UML extension mechanism.
- Verify the textual content of the class diagram with the FSDP parser that has been generated through ANTLR tool.
- Stores the detail design diagram related information in XML file.
- Retrieves the graphical representation of the system design from the stored XML file at any point of time.
- Generates possible java codes from the XML file.

Hence the reusability which is the fundamental concept of using design patterns can be achieved as pattern users can retrieve the previously created design pattern at any time and view it on the screen using this tool together with the provision of generating java codes which make implementation faster.

Verify the textual representation of class diagram

Let us now discuss on how the simulator tool works. Let us consider again the system design depicted in Fig. 2. The simulator tool has the facility to draw the system diagram similar to Fig. 2. After giving the textual information as it is there in Fig. 2 when the Save option is selected the GUI tool then verifies all the textual contents with the already generated FSDP parser. For example let us consider DB2ConnectionPool class (see Table 5). To verify if the class is defined properly following the FSDP grammar or not the tool calls the classDecl() method of FSDPParser.java. The classDecl() method uses the string "**DB2ConnectionPool { AbstractFactory/ConcreteFactory, Singleton[2]/ Singleton }**" and parses it according to the production rule. If the return result of the classDecl() method call is successful then the GUI tool goes ahead and calls the next parsing method instance(). But as there is no instance declared for DB2ConnectionPool class no tokens get generated and hence no action is taken by the tool. The tool will next call the validMethod() of the parser with the string "**createConnection(){AbstractFactory/ createProduct, Singleton[2] / Instance }**"and checks if the declaration of the method createConnection() is as per the FSDP language or not. If the parser does not throw any exception that means the class DB2ConnectionPool is specified as per the FSDP language.

Following the same procedure the GUI simulator tool checks all the textual entries of the other classes like ConnectionPool, OracleConnectionPool, OracleConnection, DB2Connection and Connection, their methods and attributes whether they are really written as per FSDP language. Finally when the tool finds no error it then gives user a choice to save the design information in a XML file and the user can save it by providing a name of the file.

Generation of XML file

Let us now provide a brief detail of the XML file which is being stored in the system with all the required information of the system design. Table 6 describes the DTD which is being used for generating the XML file by the tool.

Table 6: DTD of the XML file

```
<!ELEMENT DOCUMENT (CLASS* , RELATIONS*, LINE*)>
<!ELEMENT CLASS (RECTANGLE, NAME, ROLE, TEXT)>
<!ELEMENT RECTANGLE (#PCDATA)>
<!ELEMENT NAME (#PCDATA)>
<!ELEMENT ROLE (#PCDATA)>
<!ELEMENT TEXT (#PCDATA)>
<!ELEMENT RELATIONS (#PCDATA)>
<!ELEMENT LINE (#PCDATA)>
<!ATTLIST RECTANGLE
    X1 NMTOKEN #REQUIRED
    Y1 NMTOKEN #REQUIRED>
<!ATTLIST TEXT
    X1 NMTOKEN #REQUIRED
    Y1 NMTOKEN #REQUIRED
    CONTENT CDATA #REQUIRED>
<!ATTLIST RELATIONS
    RELATION CDATA #REQUIRED
    SOURCE CDATA #REQUIRED
    DESTINATION CDATA #REQUIRED>
<!ATTLIST LINE
    X1 NMTOKEN #REQUIRED
    Y1 NMTOKEN #REQUIRED
    X2 NMTOKEN #REQUIRED
    Y2 NMTOKEN #REQUIRED>
```

The following is a part of the XML file which represents the structural information of the OracleConnectionPool class.

```
<CLASS>
    <RECTANGLE X1="410" Y1="575" />
    <NAME>OracleConnectionPool</NAME>
    <ROLE>AbstractFactory/
            ConcreteFactory, Singleton[1]/Singleton
    </ROLE>
      <TEXT X1="86" Y1="240"
```

```
                CONTENT="OracleConnectionPool{AbstractFactory/
                ConcreteFactory, Singleton[1]/Singleton } $
                createConnection(){AbstractFactory/createProduct,
                Singleton[1]/Instance} $"
        />
    </CLASS>
```

Each class is defined within a separate node called "CLASS". The XML file for this system contains a total of 6 CLASS nodes where each CLASS node defines the detail textual information of each of the classes. The CLASS node has other child nodes like RECTANGLE, NAME, ROLE, TEXT. Child node RECTANGLE captures the pixel positions where the class has been drawn on the screen. The child node NAME represents the name of the class whereas the node ROLE declares the role(s) that particular class performs under design pattern(s). For example the OracleConnectionPool class plays the role of ConcreteFactory under AbstractFactory design pattern as well as the role of Singleton under the first instance of Singleton design pattern. These two roles are separated by a comma. The attribute CONTENT within the child node TEXT holds all the textual information provided within the class diagram. The texts within the CONTENT tag are separated by a "$" symbol which differentiates the class declaration and all the method declarations i.e after the class name and its pattern is declared a "$" is inserted. After the declaration of each method the same "$" is placed. This is done for the ease of XML parsing. The parameters X1 and Y1 point to the starting pixel position of the text inside the class diagram. We need the position of the text as the tool also supports the reconstruction of the exact graphical representation including these classes, texts within them, behavioral relationship among the participating classes etc. from this XML file only.

The structural information of the system design is stored within the CLASS node whereas the behavioral aspects are stored within the RELATIONS node in the XML file. The behavioral representation of the system as stored in the XML file is given below.

```
<RELATIONS RELATION="Dependency"
 SOURCE="OracleConnectionPool"  DESTINATION="OracleConnection"
 />
 <RELATIONS RELATION="Dependency"
 SOURCE="DB2ConnectionPool" DESTINATION="DB2Connection" />
 <RELATIONS RELATION="Generalization"
 SOURCE="OracleConnectionPool" DESTINATION="ConnectionPool" />
  <RELATIONS RELATION="Generalization"
 SOURCE="DB2ConnectionPool" DESTINATION="ConnectionPool" />
```

```
<RELATIONS RELATION="Generalization"
SOURCE="OracleConnection" DESTINATION="Connection" />
<RELATIONS RELATION="Generalization" SOURCE="DB2Connection"
DESTINATION="Connection" />
```

The parameter RELATION in the RELATIONSHIP node represents how a class is interconnected with the other class in the design pattern. For example the following **Dependency** relationship between OracleConnectionPool and OracleConnection classes indicates that OracleConnectionPool is dependent on OracleConnection.

```
<RELATIONS RELATION="Dependency"
SOURCE="OracleConnectionPool" DESTINATION="OracleConnection"
/>
```

Similarly the **Generalization** relationship between OracleConnectionPool and ConnectionPool classes indicates OracleConnectionPool inherits the characteristics of ConnectionPool.

Generation of Java code files

The GUI tool provides user the facility to generate Java code from the stored XML file at any point of time. GUI invokes a Code Generator process which creates the java class files. But the code that gets generated will not contain any business logic. The code will only reflect the structural and behavioral information of the system design. The structural and behavioral information in the XML file is thus required to generate the appropriate java code of the system design. The tool parses the stored XML file and collects all the design information. It checks how many classes are participating in the system and how are they interconnected with the other classes. After gathering all these information it generates the codes corresponding to each class.

Let us consider OracleConnectionPool class for which the code generator wants to build the java code. The code generator parses the XML file and searches the CLASS node whose child node NAME contains the value OracleConnectionPool. After finding the proper CLASS node the code generator gets ready to prepare the java code for the class OracleConnectionPool. Code generator then parses the CONTENT parameter of the child node TEXT and gathers information for all the defined methods. In this case it finds that one method createConnection() is defined in the XML file. It retrieves the parameter declaration and the method return type and stores this information in suitable data structures. Next the code generator parses the XML file for all the RELATIONS nodes and looks for entries which consist of OracleConnectionPool as the value of the SOURCE parameter. In our case it finds two child node entries exist. In one entry the OracleConnectionPool is connected with the OracleConnection class with Dependency relationship and the second entry says that OracleConnectionPool is related to ConnectionPool class with the Generalization relationship. Now the code generator has got all the required information to build the java file for OracleConnectionPool class. The tool then generates a java file named as OracleConnectionPool.java. Considering the Dependency relationship the tool declares and creates an instance of OracleConnection class inside OracleConnectionPool.java. For example inside the OracleConnectionPool class there may be a line like the following:

```
OracleConnection theOracleConnection = new OracleConnection();
```

Similarly considering the **Generalization** relationship between OracleConnectionPool and ConnectionPool classes the OracleConnectionPool class will be declared as:

```
public class OracleConnectionPool extends ConnectionPool {
........
........
}
```

By taking all the possible structural and behavioral information from the XML file the total structure of the OracleConnectionPool class that gets generated by the code generator will be following:

```
public class OracleConnectionPool extends ConnectionPool {
        OracleConnection theOracleConnection = new  OracleConnection();
public void create connection ();
}
```

Following the same process the code generator generates java files for the rest of the classes.

9 Conclusion and Future Work

Standard UML is normally used to describe a design pattern. However, UML does not provide all the necessary pattern related information to the designers especially when patterns are combined. Several research works and UML extensions have been done for the explicit visualization of design patterns in system designs. The application of a design pattern may change the names of classes, operations, and attributes participating in this pattern to the terms of the application domain. Thus, the roles that the classes, operations, and attributes play in this pattern have lost. This pattern-related information is important to accomplish the goals of design pattern. Without explicitly representing this information, the designers are forced to communicate at the class and object level, instead of the pattern level. The design decisions and tradeoffs captured in the pattern are lost too. Jing Dong et. al [Dong03,Dong07] provided UML extension mechanism to visualize pattern information of the modeling elements (classes, methods, attributes) from UML class diagrams. Most of the existing formal languages which represent design pattern basically tend to focus only on the structural and behavioral aspects of design patterns [Gamma95,OMG09,Taibi03a,Taibi03b,Hirsh99,Mikko98,Bayle07] but they do not provide any information on the various UML extension mechanisms so as to more clearly represent design patterns. In this paper we combined the works of [Dong03, Dong07] and [Taibi03a] and proposed a formal specification language FSDP which will help formal representation of design patterns which reflects the structural and behavioral aspects of design pattern as well as the role the participating model elements plays in design pattern compositions. The grammar FSDP has been verified by the ANTLR tool. We also introduced a new designing environment, a simulator GUI (Graphical User Interface) tool based on the new formal grammar FSDP (Formal Specification of Design Pattern) which will help to draw design patterns class diagrams on the tool and verify the textual information by the FSDP parser. Hence the tool provides a formal approach to represent design pattern as per FSDP specification. The tool also supports to store the pattern information in an XML file which can be further used to retrieve back the graphical representation on the GUI. There is a provision for generating java code from the XML file which may help pattern user to accelerate the implementation from any design pattern class diagram.

The tool does not support modifying of existing system design. Pattern users can now only view the saved system design any point of time. Our future work will consist of modifying the existing design so that we can extend, enhance or delete part of the system design as part of any maintenance project. For the time being the simulator accepts only UML Class diagram to verify whether the textual content is represented as per FSDP specification. Our next work will be such that the tool accepts direct texts from pattern users as mentioned in Table 5 and draw the corresponding UML class diagram. Further, the tool is not able to generate the exact java code by taking into account the design patterns related information in which the classes participate. For example the tool does not provide any pattern related code in OracleConnectionPool.java file from which developers may get the notion that the class plays dual role one of which as a ConcreteFactory under AbstractFactory design pattern and the other role is a Singleton under the first instance of Singleton design pattern. The java classes only represent the superficial framework. Our future work will also be to modify the code generator process so that the necessary java code reflects the role of the modeling elements in each design pattern.

References

[Fonto03] Marcus Fontoura and Carlos Lucena, Extending UML to Improve the Representation of Design Patterns, Software Engineering Laboratory (LES), Computer Science Department, Pontifical Catholic University of Rio de Janei, 2003

[Dong03] Jing Dong and Sheng Yang, "Extending UML To Visualize Design Patterns In Class Diagrams". Proceedings of the Fifteenth International Conference on Software Engineering and Knowledge Engineering (SEKE), pp124-131, San Francisco Bay, California, USA, July 2003.

[Dong02] Jing Dong, UML Extensions for Design Pattern Compositions, The Journal of Object Technology (JOT), Vol. 1, No. 5, pp149-161, Nov. 2002.

[Wagne02] G. Wagner. A UML Profile for Agent-Oriented Modeling. Proceedings of the Third International Workshop on Agent.Oriented Software Engineering, Bologna, Italy, July 2002.

[Medvi02] N. Medvidovic, D. S. Rosenblum, D. F. Redmiles, and J. E. Robbins. Modeling Software Architectures in the Unified Modeling Language. ACM Transactions on Software Engineering and Methodology, 11(1):2–57, January 2002.

[Zarra01] A. Zarras, V. Issarny, C. Kloukinas, and V. K. Nguyen. Towards a Base UML Profile for Architecture Description. Proceedings of the ICSE Workshop on Architecture and UML, 2001.

[Kande00] M. M. Kande and A. Strohmeier. Towards a UML Profile for Software Architecture Descriptions. Proceedings of the Third International Conference on the Unified Modeling Language (UML), LNCS1939, Springer-Verlag, pages 513–527, October 2000.

[Fonto00] M. Fontoura, W. Pree, and B. Rumpe. UML-F: A Modeling Language for Object-Oriented Frameworks. Proceedings of the 14th European

Conference on Object- Oriented Programming (ECOOP), pages 63–82, July 2000.

[Kelle99] R. K. Keller, R. Schauer, S. Robitalille, and P. Pag´e. Pattern-Based Reverse-Engineering of Design Components. Proceedings of the 21st International Conference on Software Engineering, Los Angeles, USA, pages 226–235, May 1999.

[Vlissi96] Vlissides, Coplien and Kerth, Pattern Languages of Program Design 2 ed. Addison-Wesley, 1996.

[Gamma95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns, Elements of Reusable Object-Oriented Software. Addison-Wesley Publishing Company, 1995.

[OMG09] OMG- Unified Modeling Language version 2.2. OMG, 2009.

[Taibi03a] T. Taibi and D.C.L. Ngo, Formal Specification of Design Patterns - A Balanced Approach, Journal of Object Technology (JOT), Vol. 2, No 04, pp. 127-140, 2003.

[Taibi03b] Taibi & Ngo 2003 Formal specification of design pattern combination using BPSL, IST, Vol. 45, Issue 3, 1 March 2003, Pages 157-170.

[Eden01] Eden, A.H., and Hirshfeld, Y., Principles in formal specification of object oriented architectures, CASCON'01, 2001.

[France04] France et al., A UML-based pattern specification technique, IEEE TSE,Volume 30, Issue 3, March 2004 Pages: 193-206

[Dong07] Jing Dong, Sheng Yang, Kang Zhang, Visualizing Design Patterns in Their Applications and Compositions, IEEE TSE, Volume 33, Issue 7, July 2007 Page(s):433 – 453

[Hirsh99] Eden , A. H., Hirshfeld, Y., and Lundqvist, K., LePUS–Symbolic logic modeling of object oriented architectures: A case study, Second Nordic Workshop on Software Architecture (NOSA'99), 1999.

[Mikko98] Mikkonen, T., Formalizing design patterns, Proceedings of ICSE'98, pp. 115-124, 1998

[Bayle07] I. Bayley, Hong Zhu, Formalising Design Patterns in Predicate Logic, Software Engineering and Formal Methods, 2007. SEFM 2007. Fifth IEEE International Conference on Volume , Issue , 10-14 Sept. 2007 Page(s):25 – 36.

About the authors

Shouvik Dey is presently working as a Software Engineer in IBM India Pvt. Ltd., Kolkata, India. He is pursuing Ph.D. degree under the supervision of Professor Swapan Bhattacharya from Jadavpur University, Kolkata, India. His area of research interests are Design Patterns, Object Oriented Systems and Software Engineering. He may be reached at send2shouvik@gmail.com.

Professor Swapan Bhattacharya is presently Director, National Insititute of Technology, Durgapur, India. He also holds the position of Professor in the Dept of Computer Science & Engg., Jadavpur University, Kolkata, India. He has around 27 years of professional experience in the academic and industrial sectors. A recipient of UNESCO Young Scientist Award in 1989 and Sr. Research Fellowship from National Research Council USA during 1999-2001, Prof. Bhattacharya has worked in various countries all over the world, and has about 125 publications in various international platforms. His areas of research interest include requirements engineering, functional specifications, software testing and model-based design of semi-structured software systems. He may be reached at bswapan2000@yahoo.co.in