Formal Software Design Technique – A Pattern Based Approach

SHOUVIK DEY[1]
SWAPAN BHATTACHARYA[2]

[1]IBM India Pvt. Ltd., Kolkata -700156, India
[2]National Institute of Technology , Durgapur, 713209, India
CEP 37200-000, Lavras, MG, Brazil
[1]send2shouvik@gmail.com, [2]bswapan2000@yahoo.co.in

Abstract. Design patterns are usually modeled and documented in natural languages and visual languages, such as the Unified Modeling Language. UML does not keep track of pattern-related information when a design pattern is applied or composed with other patterns. Existing graphical notations are not able to provide complete information to the designers for specifying the role and scope of execution of the participating classes and their methods in a particular design pattern or combination of patterns. Also the existing formal specification languages for design patterns are not complete enough. They basically tend to focus on specifying structural and behavioral aspect of design patterns without taking care of the several extension mechanisms. Existing formal languages are not strong enough to provide several pattern related information like the role of a participating class or a method in combination of patterns which is indeed very important information to pattern users. This paper introduces an extension to the UML Class diagram to better represent design patterns and based on this mechanism a grammar FSDP (Formal Specification of Design Pattern) for this design specification is provided to automate the software pattern design techniques. FSDP is able to represent design pattern and combination of patterns in a more informative way compare to the existing formal languages.

Keywords: Design Patterns, Pattern combination, Role of class, Scope of execution, Remote method, UML, Class diagram.

(Received January 14, 2010 / Accepted June 23, 2010)

1. Introduction

Design patterns [13] 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 [10]. Hence there is a need to retain the pattern-related information even after the pattern is applied or composed. Here we define an extension to UML. In this extension, pattern-related information is explicit so that a design pattern can be easily identified when it is applied and composed. The extensions have been defined mainly by applying the UML built-in extensibility mechanisms, such as stereotypes, tagged values.

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 [16].

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 [18].

A number of formal specification languages have emerged to cope with the inherent shortcomings of textual and graphical descriptions. However, their main problem is lack of completeness. This is mainly because they were not originally meant to specify design patterns and have been adapted to do so, or 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 [16] though a balanced approach between the structural and behavioral aspect is provided but the approach does not provide any information like where the pattern should be used or the role(s) and scope of execution of the participating classes and their methods in a particular design pattern or when the patterns are combined. The approach is also not able to provide information on the inheritance hierarchy structure of the participating classes as well as the dependency between the classes.

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 describes the UML extension mechanisms. Section 5 shows how some well-known design patterns are represented first in standard UML and then by the proposed notation. This section also compares both representations highlighting the benefits of our approach. Section 6 discusses about a non distributed scenario and Section 7 introduces the proposed FSDP (Formal Specification of Design Pattern) language. In section 8 the proposed grammar has been defined. Section 9 illustrates the proposed grammar. Section 10 shows how the grammar can be specified using a case study and Section 11 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 [1, 8], software architecture [5, 7, 6], and agent-oriented systems [4] when the original UML is not sufficient to represent the semantic meaning of the design. Medvidovic et al. [5] applied the UML extension mechanism for modeling software architectures. They extended the UML to model software architecture in UML. Kande and Strohmeier [7] extended the UML by incorporating key abstractions in ADLs, such as connectors, components and configurations. They focus on how UML can be used for modeling architectural viewpoints. Zarras et al. [6] applied the UML extension mechanism for architecture description and provided a base UML profile for existing Architecture Description Languages (ADLs). Fontoura et al. [8] 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 [4] applied the UML extension mechanisms for agent-oriented modeling. A set of new stereotypes are defined to model agent-oriented systems. Jing Dong and Sheng Yang [2] proposed new stereotypes, tagged-values and constraints to visualize design patterns in composite design patterns.

Their work uses the UML extension mechanisms to visualize the pattern-related information hidden in a class diagram. They defined new tagged values which is useful to visualize a pattern in a distributed system.

In [14] 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 [15], 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 [16,17] 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. [19] 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.

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 [3] 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

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. An extension to the UML Class diagram is proposed which will help to better visualize the Design Pattern. We have proposed a grammar which also incorporates this extension mechanism. In this discussion we have concentrated on both distributed and non distributed systems.

4. UML Extension Mechanisms

UML is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software intensive system. It is a multi-purpose language with many notational constructs. UML provides extension mechanisms to allow the user to model software systems if the current UML technique is not semantically sufficient to express the systems. These extension mechanisms are stereotypes, tagged values, and constraints.

Stereotypes allow the definition of extensions to the UML vocabulary, denoted by <<stereotype-name>>. The base class of a stereotype can be different model elements, such as Class, Attribute, and Operation. A stereotype groups tagged values and constraints under a meaningful name. When a stereotype is branded to a model element, the semantic meaning of the tagged values and the constraints associated with the stereotype are attached to that model element implicitly. A number of possible uses of stereotypes have been classified in [9]. Tagged values extend model elements with new kinds of properties. Tagged values may be attached to a stereotype, and this association will navigate to the model element to which the stereotype is branded. Basically, the format of a tagged value is a pair of tag name and an associated value, i.e., {name: value}. The

tagged values attached to a stereotype must be compatible with the constraints of the stereotype's base class.

Constraints add new semantic restrictions to a model element. Typically constrains are written in the Object Constraint Language (OCL) [11]. Constraints attached to a stereotype imply that all model elements branded by that stereotype must obey the semantic restrictions which constraints state. Note that the constraints attached to a stereotyped model element must be compatible with the constraints of the stereotype and the base class of the model element. A profile is a stereotyped package that contains model elements that have been customized for a specific domain or purpose by extending the metamodel using stereotypes, tagged values, and constraints. A profile may specify model libraries on which it depends and the metamodel subset that it extends.

5. The Proposed Extensions

This section presents some well-known patterns using standard UML diagrams; discuss this representation, and shows how it can be enhanced by adding new elements to the underlying design notation. First we discuss an implementation of Proxy pattern which is a distributed pattern and then we discuss some non distributed design patterns. The main purpose of a generalized distributed information system is to retrieve and update information, which is also distributed. The pattern described in the following section is based on this information.

5.1 Proxy Patterns

In many embedded systems data from a single sensor is used by multiple clients who reside in a different address space (task space or processor). The naïve approach to this problem is to have each client capable of tracking down and requesting the data from the data server. This is problematic because if the characteristics of the remote server change, each client must be updated as well. The Proxy pattern solves this problem by using a local stand-in for the remote data server, called a proxy. The proxy encapsulates the information necessary to contact the real data server and get up-to-date data. Meanwhile the local clients can directly call the proxy to get the data but they remain decoupled from the remote data server. The client may link to the proxy either by calling it when they need the data, or through the implementation of callbacks. Figure 1 illustrates the structure field of the Proxy design pattern [12] using standard UML.

Keeping in mind the characteristics of a generalized distributed system as mentioned earlier, we have assumed an information server capable of retrieving and updating of distributed information. Let us also assume that the two server side methods getInfo() and setInfo() are well enough to serve our purpose. The getInfo() method retrieves distributed information from the remote server and on the other hand the setInfo() method updates any new information to the server Any client that wants to avail of these two methods, calls the server methods through the ClientProxy class which takes care of all the underlying complexities needed to connect and retrieve information from the remote server. To the Client it just creates an illusion as if the method calls are executing locally like the other local method calls. Client has its own method performOperationWithInfo() which may be used to perform some local operations after retrieving information from Server.

Figure 1 lacks several pattern related information. The first one is that there is no possible way to understand the above diagram represents which design pattern.

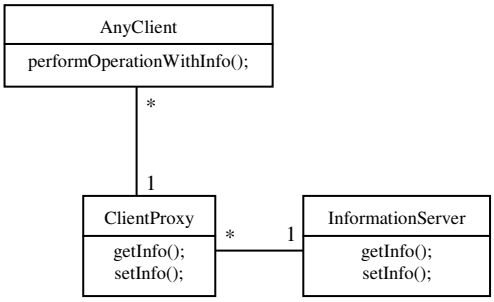

Figure 1 - Proxy Pattern.

We are saying that it is an implementation of Proxy Pattern but if only class diagram is given and no caption is there it is really hard to identify the actual design pattern we are dealing with. Moreover confusion arises when more than one pattern are composed and combined in a design diagram and some of the classes participate in more than one pattern. Next missing information is apart from the names of the classes, it is a bit difficult to designers to understand, which classes participate in the client side and that in the server side, that is the actual role of a class in a distributed design pattern is missing here. But the names of the classes may change depending on the system requirement. Also each of the methods in a class has specific job role to perform but this information is missing in this design pattern. This design pattern representation also fails to provide

information whether the participating methods reside in a distributed environment or in a standalone machine better to say whether a function executes locally or in a remote system. In a complex system where there are a large number of participating classes proper message should be conveyed to the pattern users to clearly identify these pattern related information.

To sort out these issues and to explicitly visualize design patterns in class diagrams, we define a UML extension mechanism which includes tagged-value notations. Here we introduce two new tag names and five new values of the tagged value notation, which will improve the visual representation of pattern related information. The two tag names are **role** and **scope** and the five values of the tagged-value are client, agent, server, local and remote. Role of a class actually provides information about a class under any design pattern in a system. A class may perform more than one role but that is possible when the class is participating in more than one design patterns. Whether a class resides in a client side or in the server side or it is performing other role in other design pattern, the new tag name "role" represents that. The value of the tag "role" for a class may be one of client, server or agent or any existing value. Also to resolve the first issue that is a participating class performing a specific role represents which design pattern, we have extended the tagged value notation. We propose in the way as : if a tagged value notation of a class XYZ is XYZ{role: client/Pattern} this signifies that the class XYZ performs the role of a client under design pattern "Pattern". "Pattern" points to any of the existing design pattern. On the other hand if a tagged value notation of a class XYZ is XYZ{role: agent/Pattern} this signifies that the class XYZ is in the client side of design pattern "Pattern" but it is the proxy class of the server. Similarly role of a method denotes the actual work is being done by it. Scope describes class methods whether the execution of a method is in the scope of the local system or a remote system. Its value is either local or remote.

Figure 2 shows an extended version of standard UML class diagrams of the proxy pattern using the proposed tagged-value notation. Let's take class InformationServer. The class is declared as InformationServer{role: server/Proxy} which signifies that InformationServer class plays the role of a server and this class is part of the Proxy pattern. It has two methods declared : getInfo(){role: dataRetriever}{ scope: local} and setInfo(){role: dataModifier}{ scope: local}. The getInfo() method executes locally in the server and retrieves data whereas setInfo() modifies the data within the server local space.

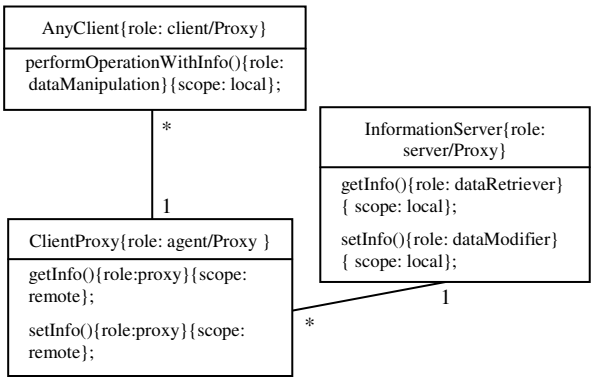

Figure 2 - Proxy Pattern with proposed Tagged-Value notation.

Similarly the getInfo() method of ClientProxy works as a proxy for the remote method and the method is not executed locally to ClientProxy class. Hence the scope is remote. It denotes that there should be some mechanism within the ClientProxy class such that this method calls a similar method residing in a remote server.

We have discussed on how design patterns can be better represented on distributed architecture by using the UML extension mechanism. Now we discuss on non distributed design patterns. In non distributed design patterns there is as such no concept of execution scope of a method because each and every method within a process executes in the local memory space during the life time of that process. Hence the scope is always local. What is important is that the role of each class playing in the particular context of the design. For overlapping patterns a class may participate in more than one role simultaneously. Also each of the methods of a class plays some specific roles. We discuss this issue and try to solve this in the following section with the proposed mechanism.

6. Example

Figure 3 shows a system design that manages the connections to different types of databases, such as Oracle and DB2. 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.

Like Figure 1, Figure 3 is also not expressive enough to provide answer to the missing pattern related information pointed in section 5.1. 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. Figure 4 gives the solution of this issue and represents the diagram using our proposed notation.

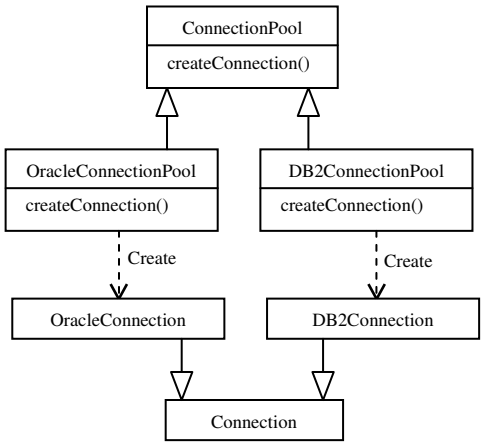

Figure 3 - Connection Pool for Database.

Let us take class OracleConnectionPool. It participates in two design patterns AbstractFactory and Singleton. Using the proposed notation the class is now expressed as OracleConnectionPool{role: ConcreteFactory/AbstractFactory, Singleton/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 Singleton pattern. Hence using this proposed notation composition of design patterns can be represented efficiently. The createConnection() method also plays dual role createProduct and Instance which is represented by using the proposed notation as createConnection(){role:createProduct, Instance}. Hence the pattern related information is not lost while patterns are composed and combined.

7. 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. For our purpose we confine our model to take English language as natural language and UML Class diagram as the visual language.

As per GoF, a properly defined design pattern should have the sections according to the following template. The template lends a uniform structure to the information, making design patterns easier to learn, compare and use [13].

Pattern Name and Classification, Intent, Also Known As, Motivation, Applicability, Structure, Participants, Collaborations, Consequences, Implementation, Sample Code, Known Uses and Related Patterns.

Though the template is used to define a single design pattern but FSDP will use the same template for combining patterns also to make the design clearer to the pattern users. In that case some of the sections may not be useful and FSDP has the ability to construct the language with keeping them blank.

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.

8. 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 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.

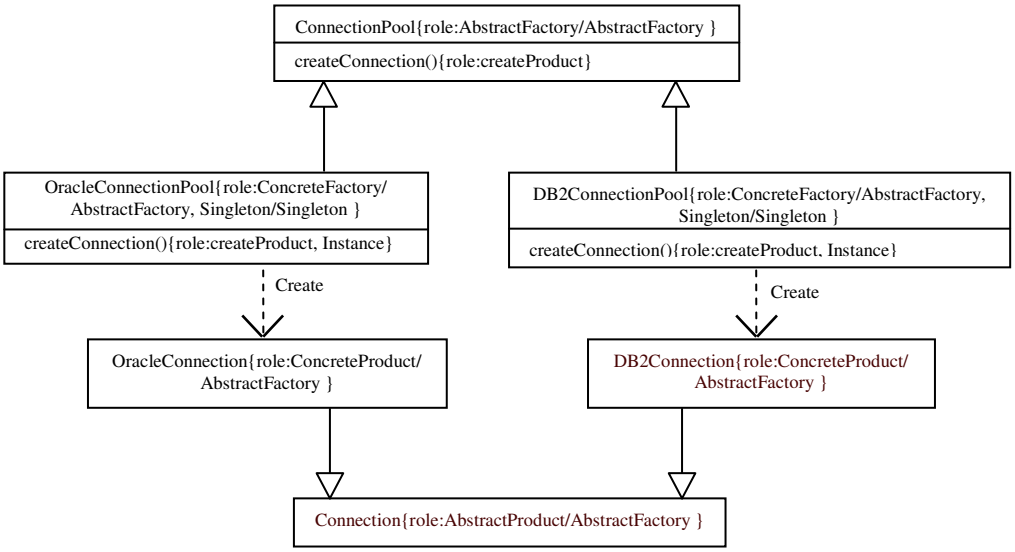

Figure 4 - Connection Pool with proposed notation.

A compiler has two parts lexer and parser. The lexers job 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 a lexer as well as the parser which is verified by the ANTLR. The character set of the proposed grammar includes the set {A-Z, a-z, 0-9} along with some special characters {. , ; : { } ( ) | _ /}

The semantics of the grammar is given below: (The terminals are in capital, non terminals in small.)

The Parser of the grammar is :

```
class FSDPParser extends Parser;
options { k=2;}
tokens {
    ROLE="role";
    SCOPE="scope";
}
validPattern : ( mandatory optional ) => mandatory optional | mandatory;
mandatory : name intent participant;
optional : ((motivation)?) => (motivation)? |((alsoKnownAs)?) => (alsoKnownAs)? | ((applicability)?) => (applicability)? | ((consequences)?) =>(consequences)? | ((implementation)?) =>(implementation)? |((samplecode)?) => (samplecode)? | ((knownUses)?) => (knownUses)? | (relatedpatterns )? ;
name : wordlist;
wordlist : WORD ;
intent : stmtlist;
stmtlist : wordlist ;
participant : structure behavior;
structure : (validClass validMethod attribute) => validClass validMethod attribute | (validClass validMethod )=> validClass validMethod | validClass;
validClass : className LEFTBRACE ROLE COLON
```

```
classRole "/" patternName (COMMA classRole "/" patternName)* RIGHTBRACE | className;
validMethod : methodDecl (roleDecl)? (scopeDecl)?;
methodDecl : methodName parameterDecl;
parameterDecl : LEFTPAREN (parameter (COMMA parameter)*)? RIGHTPAREN;
roleDecl : LEFTBRACE ROLE COLON methodRole (COMMA methodRole)* RIGHTBRACE;
scopeDecl: LEFTBRACE SCOPE COLON methodScope RIGHTBRACE;
behavior : ((dependency)?) => (dependency)? |((inheritance)?) => (inheritance)? | (association)?;
dependency : className className;
inheritance : className className;
association : className relationship className;
relationship : "one-to-many" | "many-to-one" | "many-to-many";
attribute : WORD;
className : WORD;
methodName : WORD;
methodRole : WORD;
methodScope : WORD;
classRole : WORD;
parameter : WORD ;
patternName : WORD;
motivation : stmtlist;
alsoKnownAs : stmtlist;
applicability : stmtlist;
participants : stmtlist;
collaboration : stmtlist;
consequences : stmtlist;
implementation : stmtlist;
samplecode : stmtlist;
knownUses : stmtlist;
relatedpatterns : stmtlist;
```

The Lexer part of the grammar is :

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

9. Illustration of the Language

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. 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 language is bold enough to take decision and disambiguate by looking only next two characters. The tokens section explicitly defines literals. We have defined two string literals ROLE and SCOPE which will be used repetitively in the language to specify the role of the classes and methods and the scope of execution of the methods if they participate in a distributed design. The root of the parser rule starts with validPattern. We are proposing that pattern designers have the option to use some of the sections of the template of the design pattern defined by GoF and mentioned in section 7 and some of the sections must have to be used while defining a pattern or combination of patterns. This will provide flexibility and ease to the pattern designers. Hence we have defined the root of the FSDP grammar rule as:

validPattern: (mandatory optional) => mandatory optional | mandatory;

which signifies that a valid pattern should consist of a mandatory and an optional part. The rule denotes a syntactic predicate (aka "guess" mode) which basically says first try to match both the mandatory and optional part, if it works, use it, otherwise, try the next alternative which is the mandatory part. The rule says that to define a pattern it should have at least the mandatory section. 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 mandatory rule is defined as:

mandatory : name intent participant;

which means to describe a pattern or combination of design patterns the designers must have to provide a name of the design, intent i.e. where the design would be useful and the details of the participating elements like classes, methods and attributes their structural and

behavioral aspects. Other sections of the pattern descriptions like motivation applicability are in the optional part of the grammar as these do not add any thing extra to the existing textual descriptions. Our main focus is on the solution part which is primarily defined using the structure, participants and collaboration sections. If we see further the participant rule consists of the structural and behavioral aspects.

participant : structure behavior;

The structure part of the rule holds all the necessary information about the participating classes, 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 : (validClass validMethod attribute) => validClass validMethod attribute | (validClass validMethod)=> validClass validMethod | validClass;

The structure part of the grammar rule searches for input token stream which consists of the declaration of a class, declaration of methods and attributes. If compiler finds all the three it is ok else it will back track and searches if the input token stream consists of declaration of a class and valid methods. If the result of this search is still not successful it finds for the declaration of a class only. Hence the structure part must have at least a declaration of a valid class else the input token is rejected.

The structural validClass rule is defined in the following way:

validClass : className LEFTBRACE ROLE COLON classRole "/" patternName (COMMA classRole "/" patternName)* RIGHTBRACE | className;

The rule says that a valid class can be declared by using either the proposed extension mechanism which provides the role(s) of a class under the design pattern(s) it is participating or simply mentioning only the class name. This rule thus helps to keep track of pattern related information even in a complex system when patterns are combined.

Another non terminal of the structured part is the validMethod rule.

validMethod : methodDecl (roleDecl)? (scopeDecl)?;

The rule for a valid method is defined by first declaring the method and then the specific role and scope of execution of the method under the valid class. Role and scope are declared as optional. Hence user has the choice not to mention the role or scope related information in the design pattern.

methodDecl : methodName parameterDecl ;

The methodDecl rule has two parts. First it should take the method name and then the parameter list.

parameterDecl : LEFTPAREN (parameter (COMMA parameter)*)? RIGHTPAREN;

The parameterDecl rule accepts zero parameters as well as multiple parameters separated by COMMA.

After the parameter is declared pattern users have the choice of specifying the role(s) of the method playing in the design as well as the execution scope if the pattern design is for a distributed system.

The role of the method is defined in the following way so that the information that a method may perform multiple roles can be expressed.

roleDecl : LEFTBRACE ROLE COLON methodRole (COMMA methodRole)* RIGHTBRACE;

The scope is declared as:

scopeDecl: LEFTBRACE SCOPE COLON methodScope RIGHTBRACE;

Till now we have discussed the features of the proposed grammar which takes care of the structural part of the design patterns. Now we will elaborate the other part that is the behavioral aspects of the grammar.

As already mentioned the behavior rule is defined as:

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

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 a class B is used in some methods of class A that means there is a dependency relationship exists between class A and 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. All the above behavioral interactions can be achieved by our proposed grammar. 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 association information between two classes can be achieved by the association rule.

association : className relationship className;
relationship : "one-to-many" | "many-to-one" | "many-to-many";

The first class is related to the second class by either "one-to-many" or "many-to-one" or "many-to-many" cardinality.

The optional part of the FSDP grammar is to declare the semantic rule of the sections which if not present will not cause any important information loss in representing a design pattern. This optional part of the rule consists of the sections like motivation, consequences, applicability etc. Pattern designers have the choice of declaring any number or even no one of them is required to declare

optional : ((motivation)?) => (motivation)? | ((alsoKnownAs)?) =>(alsoKnownAs)? | ((applicability)?) => (applicability)?| ((consequences)?) => (consequences)? | ((implementation)?) => (implementation)? | ((samplecode)?) => (samplecode)? | ((knownUses)?) => (knownUses)? | (relatedpatterns)? ;

10. Case Study

In this section we use FSDP to specify the pattern described in Figure 4. In section 6 we have proposed the UML extension mechanism how to increase the visualization and understandability of the design pattern representations. Now we will illustrate here how that representation can be specified in the FSDP language. For the sake of simplicity we will illustrate the structure and behavior of the design pattern as these two sections contain the major information to represent any design pattern.

The structure consists of validClass, validMethod and attribute. Let us take the ConnectionPool class and specify it using FSDP.

ConnectionPool class has methods declared but no attributes in it hence the following rule will satisfy the incoming tokens.

structure : (validClass validMethod)=> validClass validMethod

The validClass and validMethod is further specified as :

validClass: ConnectionPool {role : AbstractFactory / AbstractFactory }
validMethod : createConnection(){role:createProduct}

which is sufficient to provide the information to pattern users that ConnectionPool class participates in a single design pattern AbstractFactory and plays the role of AbstractFactory under the pattern and it consists of only one method named as createConnection and role or responsibility of which is createProduct.

Similarly the structures of the OracleConnectionPool class is specified in the following way.

OracleConnectionPool class:
validClass:OracleConnectionPool{role:ConcreteFactory / AbstractFactory,Singleton/Singleton}
validMethod:createConnection(){role:createProduct, Instance}

Note 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 Singleton pattern. Also the method createConnection() in this class performs dual role of createProduct and Instance.

Likewise the rest of the classes can be specified applying the FSDP language in the following way:

DB2ConnectionPool class:
validClass:DB2ConnectionPool{role:ConcreteFactory/AbstractFactory,Singleton/Singleton}
validMethod:createConnection(){role:createProduct, Instance}
OracleConnection class:
validClass:OracleConnection{role:ConcreteProduct/AbstractFactory }
DB2Connection class:
validClass: DB2Connection{role:ConcreteProduct/AbstractFactory }
Connection class:
validClass:Connection{role:AbstractProduct/AbstractFactory }

The behavioral aspect of the design is provided below. It shows how the classes Connection, DB2Connection, OracleConnection, DB2ConnectionPool, OracleConnectionPool and

ConnectionPool are interrelated with each other. It is clear from Figure 4 that OracleConnectionPool is dependent on OracleConnection similar to DB2ConnectionPool is dependent on DB2Connection as OracleConnectionPool creates instances of OracleConnection within it and DB2ConnectionPool creates instances of DB2Connection within DB2ConnectionPool class. This dependency relationship can be specified using the FSDP rule in the following way:

dependency : OracleConnectionPool OracleConnection
dependency : DB2ConnectionPool DB2Connection

The design pattern uses inheritance where OracleConnectionPool and DB2ConnectionPool inherit from ConnectionPool whereas OracleConnection and DB2Connection inherit from Connection class. This information can be specified using the grammar rule as:

inheritance : OracleConnectionPool ConnectionPool
inheritance : DB2ConnectionPool ConnectionPool
inheritance : OracleConnection Connection
inheritance : DB2Connection Connection

11. Conclusions 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. In this paper, we proposed a UML extension mechanism for the explicit visualization of design patterns in system designs. It is important for designers to describe explicitly patterns in a design diagram because the goals of design patterns are to reuse design experience, to improve communication within and across software development teams, to capture explicitly the design decisions made by designers, and to record design tradeoffs and design alternatives in different applications. 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. Therefore, the notations provided in this paper help on the explicit representation of design patterns and accomplishing the goals of design patterns. All the extension mechanisms are implemented in our proposed FSDP grammar. The existing formal languages to represent design pattern are not complete. They tend to focus only on the structural and behavioral aspects of design patterns but they do not support the various extension mechanisms so as to more clearly represent design patterns. The existing formal languages do not clearly provide information on how several classes are interacted with each other in terms of association, inheritance and dependency. Here we introduced a designing environment based on a new formal model, FSDP (Formal Specification of Design Pattern) which is designed to overcome these issues and aid rapid software design systems. A grammar for this design specification is provided, which has been implemented and verified by ANTLR.

The main goal of this research work is to define an adequate representation for patterns and provide a formal way of representing any design pattern using a new additional representation technique so that it may be useful in the documentation, implementation steps of the software development process. The proposed representation is complementary to existing OOADMs, and is defined an extension to UML. Our approach uses the UML extension mechanisms for visualizing design patterns. Using this new UML extension mechanism to model software system design in class diagrams, one can identify pattern-related information, such as the role of each class and its member functions, execution scope of the member methods.

This paper presented some of the well known patterns and described how their representation can be vastly enhanced with a more appropriate notation. Examples throughout the paper have shown that the approach is also valid to real world frameworks that consist of distributed design patterns.

12. References

[1] 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.

[2] Jing Dong and Sheng Yang, School of Engineering and Computer Science, University of Texas at Dallas, Richardson, “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.

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

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] S. Berner,M. Glinz, and S. Joos. A Classification of Stereotypes for Object-Oriented Modeling Languages. Proceedings of the Second International Conference on the Unified Modeling Language (UML), LNCS1723, Springer-Verlag, pages 249–264, October 1999.

[10] 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.

[11] J. B. Warmer and A. G. Kleppe. The Object ConstraintLanguage: Precise Modeling with UML. Addison-Wesley, 1998.

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

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

[14] Rik Eshuis and Roel Wieringa, A Real-Time Execution Semantics for UML Activity Diagrams, University of Twente,Department of Computer Science, Proceedings of the 4th International Conference on Fundamental Approaches to Software Engineering, Pages:76-90, 2001.

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

[16] 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.

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

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

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