Survey On Software Design-Pattern Specification Languages

SALMAN KHWAJA and MOHAMMAD ALSHAYEB, King Fahd University of Petroleum & Minerals

A design pattern is a well-defined solution to a recurrent problem. Over the years, the number of patterns and domains of design patterns have expanded, as the patterns are the experiences of the experts of the domain captured in a higher-level abstraction. This led others to work on languages for design patterns to systematically document abstraction detailed in the design pattern rather than capture algorithms and data. These design-pattern specification languages come in different flavors, targeting different aspects of design patterns. Some design-pattern specification languages tried to capture the description of the design pattern in graphical or textual format, others tried to discover design patterns in code or design diagrams, and still other design-pattern specification languages have other objectives. However, so far, no effort has been made to compare these design-pattern specification languages and identify their strengths and weaknesses. This article provides a survey and a comparison between existing design-pattern specification languages using a design-pattern specification language evaluation framework. Analysis is done by grouping the design-pattern specification languages into different categories. In addition, a brief description is provided regarding the tools available for the design-pattern specification languages. Finally, we identify some open research issues that still need to be resolved.

CCS Concepts: • **Software and its engineering** → **Specification languages**; **Design patterns**

Additional Key Words and Phrases: Design pattern specification languages, domain specific languages

ACM Reference Format:
Salman Khwaja and Mohammad Alshayeb. 2016. Survey on software design-pattern specification languages. ACM Comput. Surv. 49, 1, Article 21 (June 2016), 35 pages.
DOI: http://dx.doi.org/10.1145/2926966

1. INTRODUCTION

A design pattern is a reflection of the developer's experience and empirical knowledge, but is more generalized because it describes a problem and gives a solution to it [Alexander 1977; Lea 1994]. It provides a sound solution to a described problem. Riehle and Zullighoven [1996] defined design pattern as a recurring specific software design construct to handle a recurring issue.

Design patterns are named uniquely, but they are written in a consistent format to allow designers, developers, and others to communicate using a common vocabulary. Design patterns can expedite the design and development process of a system because they provide proven solutions to the problem, which recur commonly across multiple

This work is supported by King Fahd University of Petroleum & Minerals.
Authors' addresses: S. Khwaja, Information and Computer Science Department, King Fahd University of Petroleum & Minerals, PO Box 1172, Dhahran 31261, Saudi Arabia; M. Alshayeb (corresponding author), Information and Computer Science Department, King Fahd University of Petroleum & Minerals, PO Box 1172, Dhahran 31261, Saudi Arabia; email: alshayeb@kfupm.edu.sa.

DOI: http://dx.doi.org/10.1145/2926966

domains and architectures. Moreover, they provide ideas in a consistent high-level language.

Gabriel [1996] considered a design pattern as a constituent of expression, repetition, and configuration, in which expression is about a certain context that is repeatable but can also be resolved through a specific system configuration.

Formalizing design patterns is beneficial to enhancing the understandability of pattern semantics [Sriharsha and Reddy 2015], representing and sharing knowledge, identifying interactions [Pan and Stolterman 2013], modeling software, generating code, and detecting patterns [Shi 2007]. Nonetheless, design-pattern formalism received a lot of criticism. Pan and Stolterman [2013] reported that formalizing design patterns requires too much work to be developed; they are too formal and hard to organize; it is hard to get people to adopt them; and they are not abstract nor concrete. Seffah and Taleb [2012] also reported a few challenges with design-pattern specification languages. They indicated that pattern interrelationships are sometimes incomplete and lack context-oriented perspective.

Furthermore, Buschmann et al. [2007] indicated that there are difficulties for developers to learn the pattern languages and deal with detailed specification; thus, tool support is needed when dealing with design-pattern specification languages. They also argued that there is no definitive formal description of patterns because it must always be confined to a certain subset of options. Therefore, the required detail levels by formal approaches may restrict how generic a pattern specification language is [Buschmann et al. 2007].

The objective of this article is to present a survey and a comparison for software design-pattern specification languages. The results of this survey are beneficial to people who would like to know the pros and cons of design-pattern specification languages. The results can also be used by design-pattern practitioners, designers, and developers. In addition, in this article, we highlight different aspects of these languages to help researchers, programmers, and others in this area to find adequate information and knowledge on design-pattern specification languages.

1.1. Brief History of Design Patterns

Beck and Cunningham [1987] used some of the ideas of Alexander [1979] to develop a language for novice Smalltalk programmers based on five patterns. Coplien [2002] compiled a catalog of C++ idioms and published them as a book in 1991 [Coplien 1992]. In April 1994, the Hillside Group held the first conference on pattern languages under the title of "Pattern Languages of Programs (PLoP) Conference [1994]." Shortly thereafter, Erich Gamma et al. [1994] published their patterns book (known as the Gang of Four book).

1.2. Classification of Design Patterns

Design patterns can be classified in four different ways, which are discussed in this section.

Classification Based on Purpose/Scope: Purpose and scope are the basic criteria for this classification. The purpose criterion deals with the kind of problem that the pattern solves. The scope of the pattern is determined by the component used in the pattern. If the pattern uses classes for implementing the desired behavior, then it is a class pattern; if it uses objects to accomplish its task, then it is an object pattern [Erich Gamma 1994].

Classification Based on Intent: Patterns can be classified based on their intent. Metsker [2002] adopted the notion that the intent of a design pattern is usually expressed as the need to go beyond the ordinary facilities that are built into a programming language.

Classification Based on Relationship among Design Patterns: Another classification is based on the relationships between the design patterns [Appleton 2000]. Each pattern has a "related patterns" section in its description. Hence, patterns can be classified based on the relationship between them.

Classification Based on Organization: This classification resulted in a unique shape, which is similar to the periodic table used in chemistry. At the top level, the pattern is presented in the purest form, which is called a role model. It captures the spirit of the model without any details. The domain-specific configurations for the design pattern are added in the "type model" level. At the lowest level, the concrete deployment model is presented and is named *class model* [Lauder and Kent 1998].

1.3. Categorization of Design-Pattern Specification Languages

Design-pattern specification languages can be categorized in three different ways [Khwaja and Alshayeb 2013a] based on:

The Intent: Design-pattern specification languages can be divided into different categories according to their intention: (i) defining and describing design patterns, (ii) detection of design patterns, (iii) verification and validation of design patterns, and (iv) graphical modeling of design patterns.

The specification languages: Design-pattern specification languages can be divided into different categories according to their underlying syntax: (i) those based on mathematical formalism, (ii) those based on other modeling languages, and (iii) those based on other languages.

The notation of the language: Design-pattern specification languages can be divided into different categories according to the type of notation that they use: (i) textual notation, (ii) graphical notation, and (iii) amphibious design pattern specification languages that use both textual and graphical notation.

2. RELATED SURVEYS

Most of the research work in the pattern field focuses on microarchitectural details, that is, describing the generic aspects of software systems in an abstract fashion. It became apparent in the academic circles that, to increase the understanding of design patterns, the systematic investigation of both the system and the design pattern is necessary. Currently, work has been undertaken on the comparative analyses of design patterns; proposing precise specifications of design patterns; developing tools and rules for the recognition of design patterns, their validation, and other aspects; and identifying the relationships among patterns. Another area is the investigations of the impact of design patterns on fault-proneness of the system [Jaafar et al. 2015].

Most of the time, design-pattern application is manual. This is because the effort required to describe the specifications of design patterns is either too formal or too generic. The manual application of design patterns is difficult and highly error prone [Prechelt et al. 2001]. Precise and workable specifications can help promote the use of design patterns. Another domain in which research effort has been utilized is the identification and recovery of design patterns from the code, using the structural or behavioral aspects of the design patterns [Bernardi et al. 2014; Chihada et al. 2015; Di Noia et al. 2014; Yu et al. 2015]. Different surveys have also been conducted to identify which techniques and tools are most successful in identifying the design patterns correctly [Fulop et al. 2008; Pettersson et al. 2010; Rasool and Streitferdt 2011].

Some good research has been conducted on building a repository of design-pattern instances for practical and research usage [Chihada et al. 2015].

Understanding design patterns' microarchitectures is not given due importance [Eden 1999]. In this article, we intend to capture the "essence" of design-pattern specification languages, showing the importance of formalizing design patterns through different languages, and present a framework for categorizing these languages.

There are different types of design patterns such as architectural, requirements, and security patterns. However, in this article, we focus on specification languages of object-oriented design patterns.

2.1. Document Organization

The article is organized as follows: Section 3 discusses the existing design-pattern specification languages. Section 4 provides an overview of the framework for evaluating the design-pattern specification languages. In Section 5, we conduct an evaluation of the design-pattern specification languages, based on the evaluation framework. Section 6 presents an analysis of the evaluation for the design patterns. We present our conclusions in Section 7.

3. DESIGN-PATTERN SPECIFICATION LANGUAGES

In this section, we discuss the existing design-pattern specification languages. The languages are listed according to the underlying syntax of the design-pattern specification language. Languages based on mathematical formalism are discussed first, followed by languages based on other modelling languages, and languages based on other languages.

3.1. LePUS

LePUS is a formal approach to solving the design-pattern problem. It is very comprehensive and has been validated in the context of different design patterns; it is limited to the description of the structure of the design pattern [Gasparis 2007].

LePUS gives an abstract representation of design patterns. The specification of LePUS can be written as a formula or represented in a semantically equivalent graphical form. Both methods are sufficiently accurate and descriptive. LePUS specifications of design patterns consist of two major parts. The first part details the participants involved; the second part describes the collaboration of the participants, such as the constraints of the participants and the relationships that must or must not take place among the participants [Baroni et al. 2003].

Most LePUS relations are depicted as edges. The exceptions are the relations ReturnType and Argk, which have a more traditional, textual representation. Symbols in LePUS represent relations and (typed) variables. The formula of LePUS contains typed variables in conjunction with relation predicates. Figure 1 depicts most LePUS symbols.

The biggest restriction of LePUS is that the strong mathematical basis makes it easy for theoretical conformance but hard for use in real-world applications by software developers. There are more restrictions, as follows. Some of the design-pattern restrictions cannot be accurately expressed in mathematical expression. Some of the relationships in the design pattern, such as variants, cannot be sufficiently expressed [Kodituwakku and Bertok 2009]. The integrated tool support for it is weak. One of the tools based on LePUS is built on Prolog, and LePUS visual notation support is not offered. The visual notation of LePUS defines many abstractions to make diagrams concise. However, these elements make the diagram difficult to decipher. LePUS can only define design-pattern structures; therefore, using it in designing a system or a code [Mapelsden et al. 2002] is not well defined.

3.2. eLePUS

Attempts were made to rectify the shortcomings of LePUS by Eden et al. [1996] in eLePUS. They enhanced LePUS as a language for specifications concerning

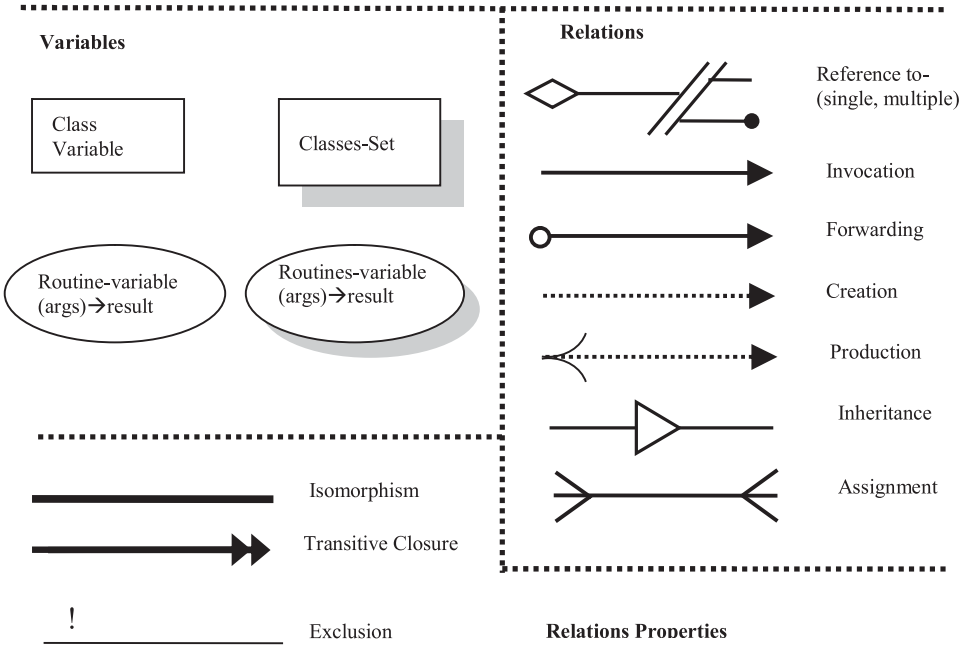

Fig. 1. LePUS symbols [Gasparis 2007].

object-oriented design and architecture. They tried to overcome the ambiguities of natural languages and the incompleteness of visual representations. Their approach was also suggested for tackling various management issues related to creating and maintaining a repository of design patterns based on its underlying mathematical model.

eLePUS provides the formalization of three additional aspects—intent, applicability, and collaboration of the design pattern—thereby augmenting the structural specifications of LePUS [Raje and Chinnasamy 2001]. The enhancements provided by eLePUS are (a) amendments to basic abstractions, (b) addition of new constructs, and (c) modifications to the representation of patterns. Moreover, eLePUS allows temporal relations, which indicate a time instance when the relation is realized.

The relationship among ground variables such as classes and functions, hierarchy variables such as inheritance, and higher-dimension variables such as a set of entities are specified in first-order logic. Three types of relationships are defined for the entities: (i) ground relations, (ii) generalized relations, and (iii) commuting relations. These relations are presented as predicates in pattern specification.

The specification problem of LePUS has not been solved completely in eLePUS. Specification of the design pattern is defined at the higher level of abstraction, but the creation of a design pattern requires much lower-level detail, such as the number of objects to create. Also, the separation of different parts of the specification makes it harder to obtain the full picture of the design pattern [Hannousse and Liu 2007].

3.3. DisCo

Mikkonen proposed the design-pattern specification language DisCo [Mikkonen 1998]. DisCo uses the concept of the layer for the formalization of the behavior of the design pattern. The refinements of these layers are used for the final description of the design pattern composition.

DisCo can also be defined as the combination of an object-oriented view with an action-oriented view. The behavioral aspect of the design pattern in DisCo is described

using the Temporal Logic of Actions (TLA) [Lamport 1994]. The necessary constituents of the formalism are (i) classes, (ii) guarded actions, and (iii) relations. The class part of DisCo is different, as it only describes data elements, and is an object and does not include any function information. Guarded actions receive objects as input, and perform data manipulation. The relations part is optional and provides transient associations among groups of objects.

The approach is successful in capturing the temporal properties of design patterns. However, this approach lacks the necessary details for implementation. In addition, the properties that the design pattern brings to the applications are omitted. Separating objects from functions violates a principal tenet of object-oriented design. Thus, the resulting specifications lack clear guidance on the structural aspect of the design pattern. Therefore, it fails to provide a good object-oriented solution.

In addition, the behavioral guidance provided by the formalism is not easy to specify using DisCo. As one action cannot invoke other actions directly, this makes action selection nondeterministic. Furthermore, there is no mechanism for imposing application-level restrictions that might be required for the correctness of the design pattern. Since the DisCo specification is not parameterized, it limits the flexibility of design patterns, but it describes both aspects of the design pattern, that is, structure and dynamics of the pattern [Martino and Esposito 2015].

3.4. Graphical Extension of BNF

Graphical Extended Backus Normal Form (GEBNF), developed by Bayley and Zhu [2007], uses the class diagram for the description of the design pattern by implementing the predicate logic.

The graphical structure in GEBNF is an extension of the BNF notation using the reference feature. To enable GEBNF, each design pattern is constrained by the first-order predicate so that the model satisfies an instance of the pattern. This makes GEBNF a meta-model language comprising an abstract syntax along with the first-order predicate. The constraints, which are based on UML, are readable and very expressive [Hedin 1998].

The constraints of the language are defined in the GEBNF definitions. Table I explains the meta-notations of GEBNF.

GEBNF specifications consist of three parts along with the identifier, which is the name of the pattern. The first section, Component, contains all the predicates that are going to be used in the design pattern specification. This makes sure that the required predicates are present for the formulae. The second portion is called Static Conditions. This portion contains the static part of the design pattern, which is evaluated using the class diagram. Thus, it defines the structural aspect of the design pattern. The final portion is called Dynamic Conditions. This portion is responsible for describing the behavioral aspect of the design pattern using the sequence and class diagrams, if needed, through the predicates defined in the first portion. GEBNF also contains consistency constraints to ensure that the diagrams are consistent [Riehle 1997].

The approach brings precision to design-pattern description, but it fails to capture the properties of generality and understandability of the design pattern [Kim and Carrington 2009]. Another limitation of the GEBNF is that it uses the predicate logic and set theory; therefore, if the developers and users are not familiar with them, then they will not be able to utilize GEBNF for the design pattern [Mattsson et al. 2009].

3.5. LOTOS

LOTOS is also a formal design-pattern specification language that is constructed on temporal ordering specification, which was proposed by Saeki [2000]. For behavioral specifications, LOTOS uses Calculus of Communicating Systems (CCS). Data is

Table I. Meanings of GEBNF Notations [Zhu and Shan 2006]

Notation	Meaning	Example and explanation
$X_1 \mid X_2 \mid \ldots \mid X_n$	Choice of $X_1, X_2, \ldots, X_n$	*ActorNode* | *UseCaseNode* means that the entity is either an actor node or use-case node
$L_1{:}X_1$ $L_2{:}X_2$ $L_k{:}X_k$	Order sequence consists of k fields of type $X_1, X_2, \ldots, X_k$ that can be accessed by the field names $L_1, L_2, \ldots, L_k$	*ClassName:Text Attributes: Attribute* Methods: Method** means that the entity consists of three parts called classname, attributes, and methods, respectively.
X^*	Repetition of *X* (include null)	*Diagram** means that the entity consists of number N of diagrams, where $N \geq 0$.
X+	Repetition of *X* (exclude null)	*Diagram*+ means that the entity consists of a number N of diagrams, where $N \geq 1$.
[*X*]	*X* is optional	[Actor]: element of actor is optional.
X	Reference to an existing element of type *X* in the model	*ClassNode* is a reference to an existing class node.
'abc'	Terminal element, the literal value of a string	*'extends'*: the literal value of the string *'extends'*.

Table II. Basic Constructors of Behavior Expressions [Saeki et al. 1993]

Operators1	Naming	Intuitive Meaning
a; *B*	Action Prefix	The event a occurs, followed by B.
$B_1 >> B_2$	Enabling Operator	B_1 first, then B_2.
$B_1 [] B_2$	Choice	Either B_1 or B_2 is executed.
$[G_1] \rightarrow B_1 [G_2] \rightarrow B_2$	Choice with guard conditions	If G_i ($i = 1, 2$), then B_i is executed.
$B_1 [> B_2$	Disabling Operator	During B_1, B_1 is discarded and B_2 is executed.
$B_1 \mid\mid\mid B_2$	Interleaving Operator	B_1 and B_2 are independently, that is, asynchronously, executed in parallel.
$B_1 \parallel B_2$	Synchronizing Operator	B_1 and B_2 are executed in parallel and synchronously with all events.
$B_1 \mid [a_1, \ldots\ldots, a_n] \mid B_2$	General Parallel Operator	B_1 and B_2 are executed in parallel and synchronously with $a_1, \ldots\ldots, a_n$.

specified using the algebra of abstract data type (ADT). LOTOS was originally created by the International Organization for Standardization (ISO) for describing and modeling the interaction of open system interconnection (OSI) layers.

The behavior specifications are the interaction sequences for the desired system. These interaction sequences are called processes. A process can be decomposed into multiple subprocesses hierarchically until it becomes an event. An event is the atomic unit of synchronized interaction, which cannot be further decomposed. The observable behavior of the process is described in the behavior expressions. Several operators are used in LOTOS for constructing behavior expressions. These constructors are listed in Table II.

LOTOS does not provide simple and clear specifications, as its strength lies in describing the network layer specifications. Saeki used LOTOS only for the example of Command and Composite pattern from the Erich Gamma [1994] design patterns. It is a very lengthy specification in LOTOS and only specifies the behavioral aspect of the design patterns [Taibi 2007].

Table III. Structure of a BPSL Formula [Taibi 2007]

$\exists$ $x_1, \ldots, x_{q1}, y_1, \ldots, y_{q2}$ $\subset C \cup V \cup M \wedge_i PR_i\ (x_j, y_k)$	$\{1 <= I <= q, 1 <= j < q_1; 1 <= k <= q_2\}$
$TR_1(z_1 < cz_1 >, t_1 < ct_1 >), \ldots, TR_m(z_m < cz_m >, t_m < ct_m >) \in$ TR; $\{z_1, \ldots, z_m, t_1, \ldots, t_m \in C\} u_1, \ldots, u_n \subset$ *(Member of C)* $\cup$ V;	
$\text{Init}_\Phi \triangleq$ P $N \triangleq A_1 \vee \ldots \vee A_r$ $U \triangleq < u_i, \ldots, u_j >$ $\Phi \triangleq \text{Init}_\Phi \wedge \Box [N]_u \wedge \text{WF}_u(A)$	{P is the initial predicate} {$A_1 \ldots A_r$ are actions} {1 <= I <= n and 1 <= j <= n} {$A = A_{i1} \ldots A_{i2}$, 1 <= i_1 < i_2 <= r}

3.6. BPSL

Balanced Pattern Specification Languages (BPSL) is another formal specification design-pattern specification language. The main objective for the development of BPSL was to complement the existing informal approaches for the design-pattern specification languages and to remove the shortcomings of the formal approaches. BPSL provides a formal structural and behavioral specification of patterns. BPSL uses layers of abstraction for the specification. These layers are pattern composition, pattern, and pattern instances [Taibi and Ngo 2003].

In BPSL, the structural description of the pattern is described in first-order logic (FOL), but the behavioral aspect of the design pattern is described in TLA. The most interesting point of the BPSL approach is the introduction of a very high abstract layer in the description of the behaviors of design patterns. Angel and Moreno-Navarro [2007] introduced temporal relations (predicates) between instances; the behavior is specified as temporal actions defined on those predicates.

The building blocks of BPSL are entities and relations. Entities (participants) include classes, attributes, methods, objects, and untyped values. The irreducible units form the primary entities. The collaboration between entities is handled by relations. Relations can be either permanent or temporal.

A temporary relation in BPSL means a relation that lasts only for a limited time then disappears, but it can appear again in the future [Taibi and Ngo 2003]. BPSL consist of three parts. The first part is SBPSL, for which S symbolizes the structural aspect of the design part, which is presented as a well-formed formula. The second part is BBSPL, for which first B symbolizes the behavioral aspect. This part consists of variables and temporal relations. The final part carries the BBPSL formula for the behavioral aspect. The structure of BPSL is shown in Table III.

The main idea of BPSL is based on LePUS and DisCo; therefore, it also shares many advantages and disadvantages of these two languages. The strength of BPSL lies in capturing the structural specifications of the design pattern. In addition, the specifications are less complex because it uses FOL, but it reduces the expressivity of the language. It has not been determined if the additional expressivity offered by LePUS helps to better capture structural properties. The abilities and limitations of BPSL in handling the behavioral properties of the design pattern are identical to those of DisCo.

Critics of the formally defined design-pattern specification languages argue that it is not clear why formal descriptions are needed. Also, the benefits of describing design pattern formally is still a questionable investment of time and effort [Henninger and Corrêa 2007].

3.7. Object Calculus

Object calculus claims to be a more appropriate formalism for reasoning about patterns because it deals directly with actions and operations as first-class elements, and with timing properties and properties of system states at general time points [Lano 2007].

A design pattern is one example of model transformation that can be achieved through object calculus. Object calculus theory contains a collection of types and symbols. There are three types of symbols that are used in the description of the design pattern. The constant symbols denote constants in the design pattern. The variables that change with time are represented through attribute symbols. Finally, the variables that are used in the operations are called action symbols. Attributes and the dynamic properties of the actions are handled through the set of axioms. Linear Temporal Logic operators are used in specifying these axioms: X (in the next state), P (in the previous state), U (strong until), S (strong since), G (always in the future), and F (sometime in the future).

The formalism in object calculus provides some advantages in regard to the rigorous reasoning and transformation of design patterns. As object calculus is textual formal language, the structural and behavioral aspects of design patterns and the relationship among design patterns are not straightforward. In addition, the language fails in providing concrete design guidelines to the developers [Aoyama 2000]. Furthermore, the requirement of a mathematics and formal logic background has made it difficult for pattern authors to adopt it for design patterns [Kim et al. 2003].

3.8. Extended Object-Oriented Programming Language Grammar

Extended Object-oriented Programming Language Grammar (EOOPLG) uses attribute extensions to specify the design patterns. EOOPLG extends the static semantics of the language to use for the design-pattern specification, but the base conventions and the syntax of the base language is retained [Hedin 1998]. The main purpose of the extensions is to handle the programming convention of the design pattern. The basis of the EOOPLG is in attribute grammars, and the semantics rules are used for describing the conventions [Hedin 1997].

The extension grammar can automatically generate a convention checker just as attribute grammar can generate an attribute evaluator. Three distinct specifications are used in this technique.

A base grammar interface: This provides the base language for the context-free grammar. All static semantic information, such as type of object or name binding, is contained in it. Functions can be used to reference other nodes in the syntax tree.

An extension grammar: To support multiple programming conventions, the extension grammar is used. The basic information is used from the base grammar interface to avoid duplicating efforts and information.

Attribute comments: To annotate an application program, special comments are used. These comments are called attribute comments: for example, /*= a =*/ is an attribute comment that defines a Boolean attribute to have the value true [Hedin 1998].

This approach is definitely unique, but it is very complex for describing compound patterns. In addition, the presence of a high number of participants in the design pattern can increase the complexity. Finally, it lacks complete specification of a design pattern [Mak et al. 2004].

3.9. Formal Specification of Design Patterns

Graphical notations are used mainly for the proper and clear description of several design patterns, and help visualize the system design. Graphical notations, such as UML class diagrams and sequence diagrams, are generally used. The Formal Specification of Design Pattern (FSDP) uses the textual content of UML class diagrams and represents it in a formal way. It also represents structural aspects, such as class methods and attributes, in a formal way as well as the behavioral nature, such as the relationships,

association, and cardinality among the participating classes [Deya and Bhattacharyab 2010].

FSDP language is described by a grammar, which determines exactly what defines a particular token and what sequence of tokens is decreed as valid. If the language being used is exactly as defined in the grammar, the parser will be able to recognize the patterns that make certain structures and group these together. 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 capitals and boldface while nonterminals are in lowercase.

No complete illustration of the software design pattern is available. An example of system-level design has been provided, but FSDP did not define the Gang of Four design pattern. The language is highly complex and lengthy. The major emphasis is on the verification of the design pattern through UML. Another goal of the language is the extension mechanism for the design pattern, based on the work of Taibi and Ngo [2003] and Jing et al. [2007].

3.10. DPML

Mapelsden et al. [2002] proposed the Design Pattern Modeling Language (DPML). DPML is a visual modeling language for design-pattern specification. DPML uses a generalization concept; it defines a metamodel of the design pattern and the instance of the design pattern is specified using a notation. In the metamodel, the logical structure of objects is described, which can be used to create an instance of a design pattern. DPML notations are diagrammatic notations, representing the model visually. DPML can represent only the generalized solution of the design pattern and not the instance of the design pattern.

The target of DPML is to provide a reasonable formalism along with a powerful representation, without being complex, which restricts the use of the design patterns in the programmer community.

In DPML, the basic pattern model is shown using specification diagrams. The notation for a specification diagram is shown in Figure 2. An abstract metamodel is the second specification diagram in DPML, and can be described as a UML class diagram.

A tailored instance of a design pattern can be created from a default instance of DPML metamodel, which is called an instance model. The default metamodel of a design pattern encompasses all objects and constraints, so that it can create any customized instance model according to the requirement of the system.

A pattern is created by binding the elements of the pattern to the elements of the UML model. The created model consists of "proxy" elements, which are instantiated from the pattern participants, and "real" elements, which are specific to the application created during the realization of the pattern. "Dimension" plays the role of the participant, which can be carried out by more than one model element.

DPML does not provide the complete design-pattern model, only a generalized solutions model [Mapelsden et al. 2002]. Therefore, DPML descriptions are at a high level of abstraction and do not contain detailed information. DPML cannot be used accurately to identify design patterns in source code. Finally, the creation of the new notation for DPML is not very clear when DPML is created for the UML models.

3.11. RBML

Kim proposed the Role-Based Metamodeling Language (RBML) for the specification of design patterns that is based on UML metamodeling technique [Kim 2007]. RBML is based on the concept of role elements, which is used by Montes and Vela [2003] in depicting pattern diagrams. The idea of *role elements* and *bonds* is not well defined in pattern diagrams. However, in RBML, visual notations are based on UML 1.4. Also, to

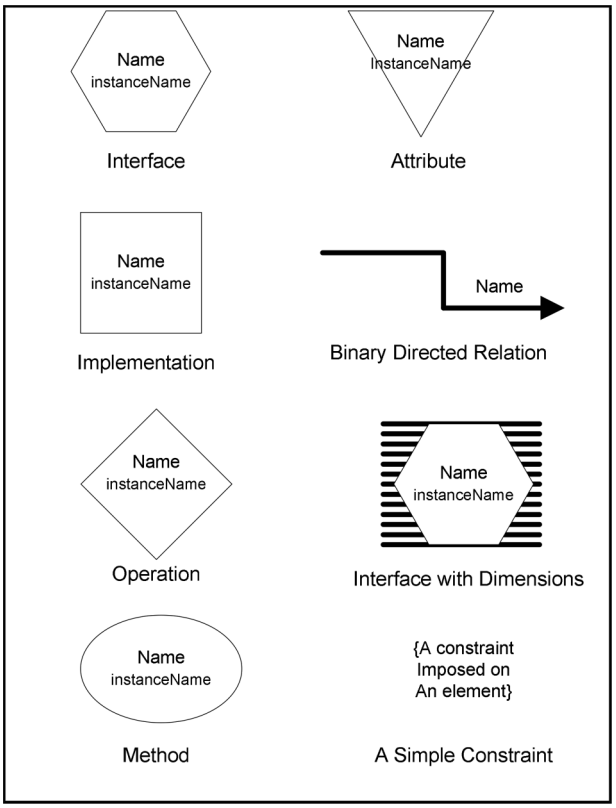

Fig. 2. Basic DPML notation [Mapelsden et al. 2002].

specify pattern properties, the Object Constraint Language (OCL) was employed [OMG 2005]. Therefore, RBML is more elaborative and addresses more aspects of solutions proposed by design patterns [Bohdanowicz 2005].

RBML uses a family of UML models for the specification of the design pattern as model roles [Kim 2007]. A model role consists of the UML metaclass, its properties, and model elements. It is linked to the role's base metaclass. A set of model roles specifies all aspects and restrictions of design-pattern specification.

The UML infrastructure is defined as a four-layer metamodel architecture: the top layer, M3, is used for the metamodel specification language. The next layer, M2, specifies the UML metamodel. UML models are specified in M1; the last layer, M0, contains the configurations of the object for the UML model defined at layer M1. Pattern roles are defined at level M2 as a specialization of a UML metaclass.

An instance of design pattern can be created from the RBML metamodel as shown in Figure 3, which defines the abstract syntax for RBML specifications. The metamodel stipulates that every pattern role must have a name, the base metaclass, and a realization multiplicity. A realization multiplicity is defined by the lower bound and upper bound of the role. In Figure 3, the subclasses of the PatternRole class are the types of structural pattern roles that define specializations of their base in the UML metamodel. The instances (roles) of the RBML metaclasses are expressed in the UML notation, thus observe the syntax of the UML.

The main drawback of the RBML is that it requires an extension of the UML metamodel with new elements. Metamodeling is a first-class extension mechanism of UML

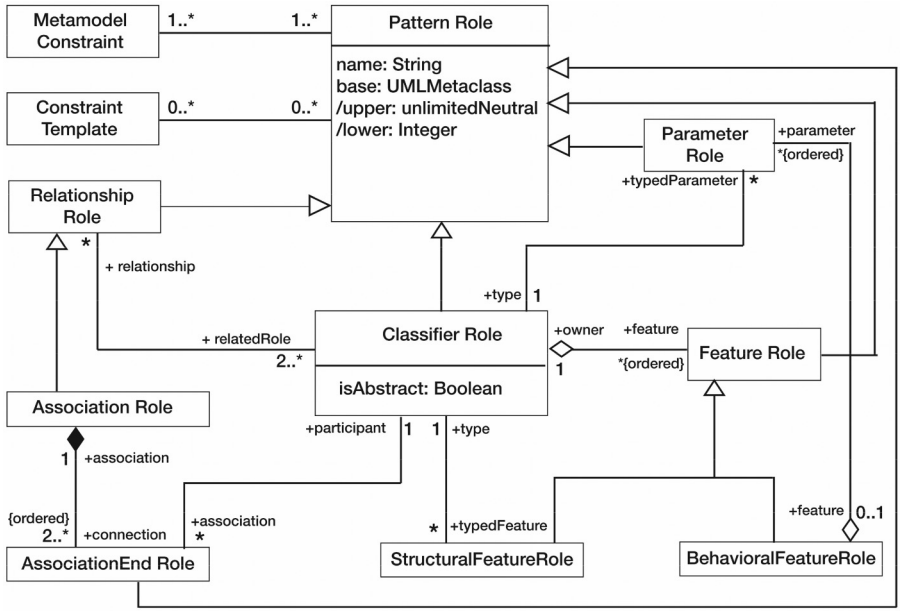

Fig. 3. A partial RBML kernel [Kim 2007].

2.0, handled through the Metaobject Facility (MOF). It provides almost unlimited possibilities of extending the UML 2.0 metamodel. Moreover, the notation for representing a design-pattern instance proposed in RBML is not clearly defined.

The major advantages of RBML are the following: it allows and supports the use of a design pattern directly through UML; it has the capability to capture various perspectives of a design pattern; its direct reliance on UML makes it easier to be supported by UML; it provides precise and concise representation of pattern properties; it provides a rigorous notion of pattern conformance for UML models; and it offers tool support that facilitates the systematic use of patterns in the development of UML models [Kim 2007].

3.12. Constraint Diagrams

This model utilizes constraint diagrams along with UML for the description of selected design patterns. This is achieved through the utilization of recent progress made in visual modeling notation to achieve greater clarity without requiring the use of obtuse mathematical symbols [Lauder and Kent 1998].

Constraint diagrams are based on a collection of sets, upon which constraints can be specified that will apply to the members of the set. Sets are shown as Venn diagrams. The members of the set are shown by arbitrary numbers of dots within or on the edge of the set. Distinct members are shown as two or more unconnected dots. A single element can exist in two or more positions and is represented by multiple dots connected with an arc. However, an element may exist at one position only at any given time [Kent 1997].

An instance of class is represented by extending the UML notation. An extension is done in class symbol of UML to hold the constraint diagram for the abstract instances of the class.

The top layer is the role model. The pattern is expressed in abstract states and abstract behavioral semantics, and forms the constraint sets. The abstract layer lacks

domain-specific details or application-specific details and depicts only the essential part of the design pattern.

The domain-specific refinements to the design pattern are contained in the second layer. The second layer is the type model. This converts the abstract semantics of the earlier layer into the concrete syntax.

The last layer is the class model. It contains the application-specific conditions by applying the attributes and method implementations on the earlier state to reach concrete states and semantics.

The spirit of the pattern is captured using the role model. It does not contain nonessential features, thus describing only the basic structure for the pattern through objects and their static and dynamic properties. State and behavioral constraints can be included in the role model.

Constraint diagrams show that visual notations can represent the spirit of design patterns precisely and clearly. The essential part of the pattern is defined in a role model. Type models and class models are used to further refine and implement the pattern. Three-layer modeling is used to achieve abstraction without losing clarity and generality in describing the pattern. The easier interpretation of the formal specification of design patterns is also one of the goals of this language.

The notation of constraint diagrams is difficult because the differences between diagrams at different levels are not very clear and can be hard to understand [Mapelsden et al. 2002].

3.13. Design-Pattern Definition Language

XML is a markup language that is used to represent information as semistructured data [W3C 2008]. XML documents cannot be processed if they are not well formed. The Design-Pattern Definition Language (DPDL) [Khwaja and Alshayeb 2013b] is based on XML. XML provides flexibility, simplicity, and is quite common in the computing world. The XML schema of the DPDL language is shown in Figure 4. DPDL divides the schema based on the structural and behavioral aspects of the design pattern. The constituents of the design pattern are grouped in the structuralAttributes element of the DPDL schema. The interactions of these constituents of the design pattern are defined in the behavioralAttributes element of the schema. There is another element *ForFuture* in the schema, which is left for extending the schema in the future.

3.14. RSL

Sterritt et al. [2010] designed RSL for a formal specification of the GoF pattern, which is based on the RAISE language. The main objective of the language is the modeling and verification of the design patterns using a tool developed specifically for Java. A tool DePMoVe is also created with the design-pattern specification language for pattern modeling and verification.

RSL is a formal model for specifying design patterns. The RSL model is subdivided into 7 different types of schemas. These schemas and their grouping is the construct of all design patterns. The schemes also satisfy the well-formedness condition. Table IV list the seven main RSL schemas.

3.15. SPINE

SPINE is loosely based on Prolog, as the HEDGEHOG proof engine uses an internal proof system similar to Prolog's execution [Blewitt 2007]. Pattern definitions are considered declarative by nature. Prolog is a declarative language; thus, Prolog is a nice fit for pattern definitions.

Patterns are defined in terms of a number of standard predicates that correspond to the structural and semantic constraints. For example, structural predicates include

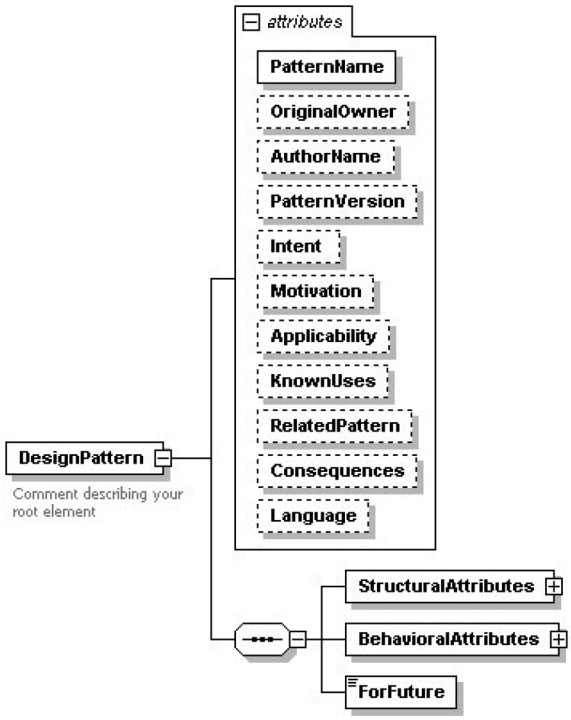

Fig. 4. DPDL high-level schema [Khwaja and Alshayeb 2013b].

Table IV. RSL Schemas as the Formal Basis [Flores et al. 2007]

Scheme	Description	Identifier
Types	General definitions of the model.	G
Methods	Operations or methods that form a class interface.	M
Classes	Structure and behavior of classes in an OO design.	C
Relations	Set of valid relations that link classes on a design.	R
Design_Structure	Consistent link between classes and their interrelations.	DS
Renaming	Correspondence between names from design to those from a pattern, that is, setting of pattern roles played by design entities.	DR
Design_Pattern	Set of generic functions that sum the previous ones and help to formally describe any design pattern.	

isAbstract(C) and typeOf(M). The arguments for these predicates are literals that identify the elements of the source code; for the sake of simplicity, references to Java classes and methods adopt the JavaDoc notation com.Example#method(type). Thus, isAbstract('com.Example') is true when com.Example is an abstract type.

These can be joined with standard connectives, such as "*and,*" "*or,*" and implied to form logical statements over a range of classes and methods. As a result, it is possible to be very specific that a particular class has some combination of methods or field types. It is also possible to specify a constraint that exists over a range of classes. The two quantifiers "*forAll*" and "*exists*" can be used to iterate over set operators, such as methodsOf and subclassesOf (or even literal lists of classes). For example, and([isAbstract(C),forAll(subclassesOf(C),Cs. isFinal(Cs)]) declares that both C is an

Table V. SLAM-SL Quantifiers [Herranz et al. 2002]

Symbol	Generalizes
Exists	Λ with false
exists1	As exists, but limiting the count to 1
Forall	V with true
Sum	+ with 0
Prod	x with 1
Count	inc with 0 (counting!)
Select	Searching
Max	Max
Maxim	Maximizers
Filter	Filters
Map	Apply a function to every element in a collection

abstract class and all of its subclasses (Cs) are final. At evaluation time, the *forAll()* is expanded into a conjunction([isFinal(Cs), . . .,isFinal(Csn)]) [Blewitt 2007].

Together, these statements can be used to define certain properties of classes. This technique works for any statements about class implementation, though, so far, it has only been used to reason about patterns.

HEDGEHOG is used for interoperating the pattern specified using SPINE. It allows the user to specify relationships between classes and path-insensitive analysis of the semantics. Some complicated semantic analysis is hard-wired to the built-in capabilities of HEDGEHOG predicates. This means that SPINE is, to some extent, bounded by the limitations of HEDGEHOG [Kumar and Kumar 2010].

The main purpose of SPINE is the automatic verification of design patterns in Java. It also has low precision in detecting design patterns. As design patterns are defined as constraints on the Java language, it is not as generic as the design patterns are and is limited only to Java. In addition, instead of capturing the essence and underlying concept of the design pattern, SPINE tries to capture the behavioral aspect of the design pattern to identify it in the code. This approach may be good for detecting a design pattern, but it is certainly not good for representing or implementing a design pattern.

3.16. SLAM-SL

Herranz et al. [2002] used the reflective features of the SLAM language for the formalization of design patterns. It is an object-oriented, formal specification language. A development environment is also provided to generate a readable code for programming languages such as Java and C++. It also integrates algebraic specifications and model-based specifications.

The toolkit of SLAM-SL is based on existing types from Booleans to tuples and collections, which are part of normal programming language. The syntax is a combination of type with its values. A sequence contains the type along with the value, for example, [integer] [1, 2], in which the first part of the sequence indicates the type and the second part indicates the value. Similarly, the values of tuple of type (Char, Integer) can be written as (‘a’, 32), and so on.

Some of the predefined quantifiers of SLAM-SL with a description are given in Table V.

SLAM-SL is created for developers who are fluent in object-oriented programming languages. Its syntax and semantics are easy for object-oriented programmers. The main purpose was to obtain a code in high-level programming language such as Java, which can then be edited and improved by a programmer. This reduced the capability of expressing logic in the language itself [Herranz and Moreno-Navarro 2003].

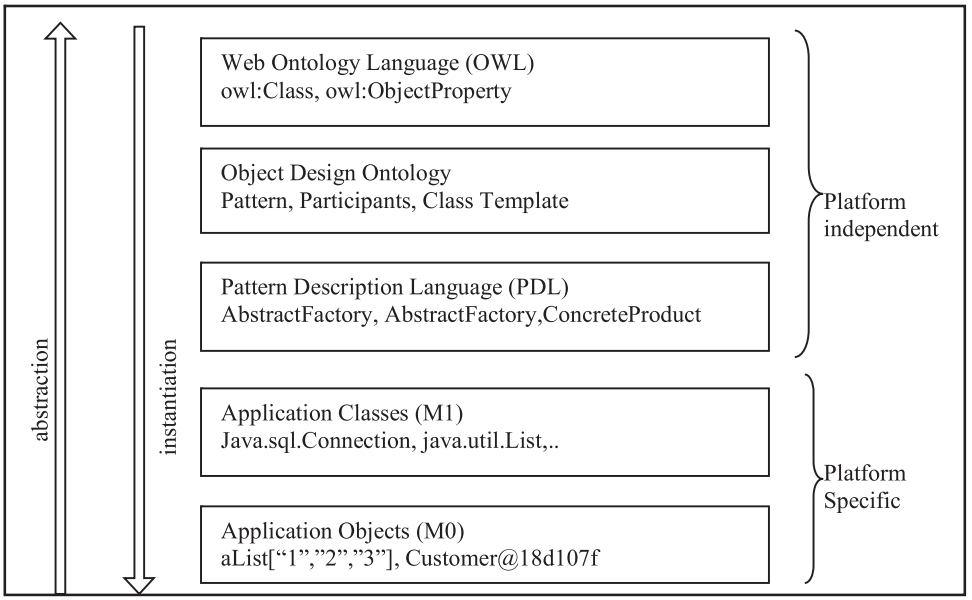

Fig. 5. Metamodeling architecture of OWL [Dietrich and Elgar 2007].

3.17. ODOL (OWL)

The W3C Web Ontology Language (OWL) [2004] is used to describe vocabularies for resources. OWL is based on the open-world assumption, which is used for pattern descriptions. The second component of ODOL is RDF (resource description framework) [2004]. RDF is used to define relationships between different resources, which are identified by URIs. Both OWL [2004] and RDF [2004] are standards of the W3C [2004] semantic web initiative. There are special constructs in OWL, which can be used to merge or disjoin design pattern models.

In OWL, the design patterns, their participants, and the properties of and relationships between the participants are based on OWL ontology, which is based on the system of OWL classes, their properties, and relationships. The proposed metamodel is shown in Figure 5.

The lower layers correspond to MOF M0 (application objects) and M1 (application classes, members, and associations). M1 artifacts instantiate the pattern participants defined using the pattern description language (PDL). These are the variables found in pattern definitions such as the AbstractFactory and the AbstractProduct in the AbstractFactory [Erich Gamma 1994] pattern. These variables are types. The types are constraints restricting the kind of (M1) artifact that can instantiate these variables. These types are modeled in the object design ontology layer (ODOL). In ODOL, the (OWL) ontology defined is the base for the pattern definitions. It contains classes such as ClassTemplate and MethodTemplate and their relationships, as well as the Pattern class representing patterns themselves. The metamodel of ODOL is OWL—ODOL contains instances of OWL classes (e.g., owl:Class and owl:ObjectProperty).

In particular, object properties are associated with OWL property types. For these types, the OWL semantics defines rules that can be used by reasoners to check the consistency of the model, and to infer additional assertions. Table VI shows some of the built-in rules.

Table VI. ODOL Rules (Selection) [Dietrich and Elgar 2007]

Property	Domain	Range	Rule(s)
isSubPatternOf	Pattern	Pattern	transitive
contains	ClassTemplate	MemberTemplate	inverseOf "owner"
owner	MemberTemplate	ClassTemplate	inverseOf "contains," functional
associationClient	AssociationTemplate	ClassTemplate	functional, inverseOf "isClientIn"
associationSupplier	AssociationTemplate	ClassTemplate	functional
isClientIn	ClassTemplate	AssociationTemplate	inverseOf "associationClient"
isSubclassOf	ClassTemplate	ClassTemplate	transitive
overrides	MethodTemplate	MethodTemplate	transitive

In addition to these rules, the ontology contains references to informal definitions of some of the concepts defined. ODOL contains formal semantics intended for machine processing as well as informal semantics intended for human processing.

3.18. URN Design Pattern Formalizing Techniques

The User Requirements Notation (URN) was developed to model and evaluate the requirements with goals and scenarios by the International Telecommunication Union in 2008. URN is built on Goal-oriented Requirement Language (GRL) for modeling agents and intentions. For describing scenarios and architectures, another notation—Use Case Maps (UCM)—is used [Roy et al. 2006].

Pattern formalization through URN is based on trade-off analysis using the solution description. For design-pattern formalization, URN is used along with the GRL. GRL is used for the description of the behavioral aspect of the design pattern. GRL is used for handling the nonfunctional requirements of the design pattern. URN and GRL are used in combination [Itu-T 2002].

The second language of URN, UCM, handles the flow of responsibilities in a scenario, which is then mapped on the structure of components. UCM represents the activities (i.e., operations, tasks, functions, and so on) to be performed or the responsibilities to be dispatched. These activities can be performed on software constructs (e.g., objects, databases, servers) and also on nonsoftware constructs such as actors or hardware resources [Itu-T 2002].

Pattern representation uses GRL to add an explicit model of the forces addressed by a pattern and the rationale behind them. It also allows the relationships between patterns to be modeled.

Patterns are modeled as tasks (hexagons), which are ways of achieving a goal or a soft goal. The nodes of this goal graph can be connected by different types of links. The direct contributions of a pattern on the goals are shown as straight lines. Side effects (indirect contributions called correlations) are shown as dotted lines. Contributions can be labeled with a "+" (the default value if none is present) or "–" to indicate that they are positive or negative.

A main objective of URN is to help designers select the appropriate design-pattern specification language for their application. Therefore, it lacks the syntax to comprehensively define the implementation aspects of the design pattern.

3.19. Layered Object Model

The layered object model (LayOM) is a custom-built language for design-pattern representation in programming languages. LayOM is an object-oriented language with special components to cater to the needs of design patterns, such as states, categories, and layers [Bosch 1996a, b].

An object in LayOM contains instance variables and methods. The semantics also match conventional object models. The difference between conventional and LayOM objects is that, in a LayOM object, the instance variable can encapsulate a layer for additional functionality to handle the state of the variable. The default state of the object is a concrete state from which an externally visible abstract state of an object can also be created. The abstract state is a simpler version of the state with a lower number of dimensions and domains. An expression of the client category provides the subset of the clients that can be handled by the class.

The behavioral aspect of design patterns is handled by the categories. LayOM classifies relations into three categories: structural, behavioral, and application-domain relations. The structural relation types are applicable on the structure of the class and their reusability. The concept of inheritance and delegation, which the structure of the class uses, are examples of the structure relation types. The behavioral relations are used to depict the relations between an object and its clients. The object uses the methods of the class; different objects can obtain different outputs from the same method. Therefore, the behavioral relations depend on the object and restrict the behavior of the class, based on the type of the object. The application-domain relation is the last relation type that handles the cases in which reusability can be achieved at the application-domain level; for example, control relation is a domain-level relation for the process control.

The special components of LayOM—such as layers, categories, and states—provide some useful and ingenious ways to define the roles of different actors in the design pattern. The solution of LayOM for design-pattern specification is elegant, concise, and comprehensive, and preserves their essence and spirit. Unfortunately, there is no readily available specification for design patterns available in the literature. Also, the specifications are highly specialized, and no generic solution has been devised as yet [Eden et al. 1997].

3.20. Abstract Data View

An Abstract Data View (ADV) is a formal model for pattern definition and application. ADV divides designs into both objects and views in order to maintain a separation of concerns. ADV provides a systematic design method that is independent of specific application environments. It allows the specification for the steps in pattern instantiation unambiguously [Alencar et al. 1995].

An Abstract Data Object (ADO) is an object responsible for an application that is closed and is not linked directly with other entities. ADO provides a public interface, which can be used to check the state of the object and to update the state of the object. ADV extends ADO by providing a user interface to a view or a public interface to the ADO, thus changing the way an ADO is viewed by other ADOs. A view may change the state of an associated ADO through an input action (event) in a user interface or through the action of another ADO.

Since an ADV is conceived to be separate from an ADO, yet specifies a view of an ADO, the ADV should incorporate a formal association with its corresponding ADO. An ADV knows the name of any ADO to which it is connected, but an ADO does not know the name of its attached ADVs. Due to the separation between the view and the object, it is possible to use several ADVs to create different views for a single collection of ADOs.

Components of the ADV model are called objects since their schema specifications describe behavior over the lifetime of the object, and involve both static and dynamic properties. Both interface and application components are composed of dynamic properties that specify changes in the attributes representing the state memory of the objects.

```
ADV ADV_Name For ADO_Client_Name; ADO_Component_Name
   Declarations
      Data Signatures
           . . .
      Attributes
           . . .
      Effectual Actions
                  Action_1 For ADO_Component_Name;
                  . . .
                  Action_N For ADO_Component_Name;
   Dynamic Properties
      Interconnections
           . . .
      Valuation
           . . .
      Behavior
           . . .
End ADV_Name
```

Fig. 6. An ADV schema for a view [Alencar et al. 1995].

```
Class of ADV ADV_Name
   Declarations
      Data Signatures      - sorts and functions
      Attributes           - there is an attribute to control object instances
      Actions              - actions to manage the creation and destruction process of
objects
      Nested ADV - contains a list of references
   Static Properties
      Constraints  - constraints in class attributes values
      Derived Attributes- non-primitive class attribute descriptions
   Dynamic Properties
      Interconnections     - interconnection with ADV specification actions
      Valuation    - valuation properties of creation and destruction actions
      Behavior             - describe the sequencing of actions
End Class
```

Fig. 7. A descriptive schema for ADV [Alencar et al. 1995].

ADVs and ADOs have distinct roles in a software system; as a consequence, they are described by different schemas [Alencar et al. 1995], as shown in Figures 6 and 7.

ADV is built on the idea of separation of action from the definition. This clear separation is believed to provide easier maintenance and understandability of the design pattern. This philosophy makes it difficult to handle the diverse possible relations between constructs of OOPLs [Eden et al. 1997].

4. EVALUATION CRITERIA

In this article, we adopt the design-pattern specification language evaluation framework that we proposed earlier in Khwaja and Alshayeb [2013a]. The framework is divided into two parts: basic properties and core properties, which are further subdivided. Table VII lists the properties of the evaluation framework. The details of each evaluation criterion can be found in Khwaja and Alshayeb [2013a].

Table VII. Evaluation Framework for Design-Pattern Specification Languages

1. Basic Properties	
Ease of Use	• Basis • Learning curve for programmers • Target
Support	• Platform independence • Integratable in IDEs • Template support • UML support • Graphical support
2. Core Properties	
Conciseness	• Capability to identify participants distinctly • Capability to identify the structure distinctly • Capability to identify collaborations distinctly
Formalism	• Original formalism • Textual formalism provided by the language
Comprehensiveness	• Ability to handle multiple entities • Ability to address object-oriented paradigm • Capability to formalize OO patterns • Other features
Expressiveness	• Graphical notation provided by the language • Supported views • Textual notation provided by the language

In addition to the evaluation framework properties that we identified in Khwaja and Alshayeb [2013a], we add another category of properties: tool capabilities.

5. DESIGN-PATTERN SPECIFICATION LANGUAGE EVALUATION

In this section, we evaluate the surveyed design-pattern specification languages using the framework presented in Section 4 [Khwaja and Alshayeb 2013a].

5.1. Basic Properties Comparison

Table VIII presents the evaluation of the basic properties of the languages.

5.2. Core Properties Comparison

Tables IX and X show the evaluation of the core properties of the surveyed design-pattern specification languages.

5.3. Tool Capabilities of the Languages

We add another category of properties: tool capabilities. In the following section, we present these properties, then conduct an evaluation of these tool features for all the surveyed design-pattern specification languages.

Tool name: Name of the tool, if it exists.

Ability to create design pattern: This feature enables a user to create design patterns. Some tools enable a user only to display the design pattern.

Ability to verify the design pattern: This feature enables a user to validate and verify the design-pattern instance according to some proven design-pattern template.

Ability to detect the design pattern: This feature enables a user to use the tool to find and detect a design pattern used in the source code of the software, as not all design-pattern specification languages are designed with the intention of detecting design patterns from the source code.

Table VIII. Evaluation of the Basic Properties of Design-Pattern Specification Languages—Part A

	Ease of Use		
Language Name	**Basis**	**Learning curve**	**Target**
Languages Based on Mathematical Formalism			
LePUS	Mathematical Logic	High	Verification of design pattern on first-order logic basis
eLePUS	Mathematical Logic	High	Design pattern designing through mathematical formalism
DisCo	Temporal Logic of Action	High	Capturing behavioral aspects of design patterns
GEBNF	Mathematical Logic	Medium	Capturing structural and behavioral aspects of design patterns in a well-structured format
LOTOS	Temporal Logic of Action	High	Verifying the behavioral aspect of design patterns
BPSL	Temporal Logic of Action	High	Capturing behavioral aspect of design patterns
Object-Calculus	Mathematical Logic (using axioms)	High	To prove design pattern as refinement transformation
EOOPLG	Attribute Grammar	High	Automatic checking of design patterns for correctness
FSDP	Mathematical Logic	Medium	To specify and recognize design patterns from the UML class diagram
Languages Based on Existing Modeling Languages			
DPML	UML based	Medium	Creating design patterns in UML
RBML	UML based	Medium	Adding support of design patterns in UML
Constraint Diagram	Constraint Diagrams + UML	Medium	A precise visual specification of design patterns
DPDL	XML	Low	Easy initiation and implementation in software development
Languages Based on Other Languages			
RSL	Based on RAISE language	High	For checking design correctness and pattern-based modelling
SPINE	Prolog	High	Verification of design pattern implementation in applications
SLAM-SL	Based on SLAM language	Medium	To use the reflection capabilities of SLAM language to specify design patterns
ODOL (OWL)	RDF and OWL	Medium	Open and extensible description of design patterns to facilitate the sharing of knowledge
URN	GRL + UCM	High	For modelling and analyzing design patterns with goals and scenarios
LayOM	New object-oriented language	Medium	For explicit representation of design patterns in programming language
ADV	ADV+ADO	High	To clearly specify and formally separate interface from the design patterns

Ability to visualize the design pattern: This feature enables a user to convert the design pattern to any visual form.

Ability to save the design pattern: This feature enables a user to save an instance or a template of the design pattern so that it can be opened later in the same state as it was before it was saved.

Table VIII. Evaluation of the Basic Properties of Design Pattern Specification Languages—Part B

	Support				
Language Name	**Platform Independence**	**Integratable in IDEs**	**Template Support**	**UML Support**	**Graphical Support**
Languages Based on Mathematical Formalism					
LePUS	NA	No	Yes	No	No
eLePUS	NA	No	Yes	No	No
DisCo	NA	No	Yes	No	No
GEBNF	NA	No	Yes	Yes	Yes (Optional)
LOTOS	NA	No	No	No	No
BPSL	NA	No	Yes	No	No
Object-Calculus	Yes	No	Yes	No	No
EOOPLG	NA	No	Yes	No	No
FSDP	NA	Partial	Yes	Yes	Yes
Languages Based on Existing Modeling Languages					
DPML	Yes	No	Yes	Yes	UML
RBML	Yes	No	Yes	Yes	UML
Constraint Diagram	NA	No	Yes	Yes	Yes
DPDL	Yes	Yes (using XML)	Yes	Yes	Yes (Optional)
Languages Based on Other Languages					
RSL	NA	No	Yes	No	Yes (Optional)
SPINE	NA	No	Yes	No	No
SLAM-SL	No	No	No	No	No
ODOL (OWL)	No	No	Yes	Yes	Yes (Optional)
URN	No	No	No	No	Yes
LayOM	No	Partial	Yes	No	No
ADV	No	No	No	No	Yes

Ability to open the design pattern: This feature enables a user to open the saved instance of a design pattern in the same state as it was at the time of saving.

Save Format: This feature enables a user to identify the format that the tool accepts for saving and opening a design pattern file.

Tool purpose: This feature explains the main purpose for creating the design-pattern tool. The evaluation of the tool features is shown in Table XI.

6. DISCUSSION

In this section, we present an analysis of the comparison results.

6.1. Basic Analysis

6.1.1. Capability to Identify Participants Distinctly. This property enables individual units to be identified separately in the design-pattern specification language. All languages, except those belonging to languages based on mathematical formalism, can identify design pattern participants individually, except for object calculus. Object calculus uses collections with the action symbol, which makes it hard to identify all individual participants distinctly. Not all languages based on existing models have this capability. The DPDL and Constraint Diagram have the capability to identify participants distinctly. The DPDL uses XML when one group is for the objects of design patterns. In the Constraint Diagram, sets are used in a Venn diagram, in which each member is shown as a dot. Languages based on other languages are also mostly not capable of identifying participants distinctly, the only exceptions being LayOM and SLAM-SL. Both LayOM

Table IX. Evaluation of the Core Properties of Design-Pattern Specification Languages—Part A

	Conciseness			Formalism	
Design Pattern	**Capability to identify participants distinctly**	**Capability to identify structure distinctly**	**Capability to identify collaborations distinctly**	**Original formalism**	**Textual formalism**
Languages Based on Mathematical Formalism					
LePUS	Yes	No	Yes	Yes	Yes
eLePUS	Yes	No	Yes	No	Yes
DisCo	Yes	No	Yes	Yes	Yes
GEBNF	Yes	Yes	Yes	No	Yes
LOTOS	No	Yes	No	Yes	Yes
BPSL	Yes	No	Yes	Yes	Yes
Object-Calculus	No	No	No	Yes	Yes
EOOPLG	Yes	Yes	Yes	No	Yes
FSDP	No	No	No	Yes	Yes
Languages Based on Existing Modeling Languages					
DPML	No	Yes	Yes	No	Yes
RBML	No	Yes	Yes	No	No
Constraint Diagram	Yes	Yes	Yes	No	No
DPDL	Yes	Yes	Yes	No	Yes
Languages Based on Other Languages					
RSL	Partial	Yes	Yes	Yes	Yes
SPINE	No	No	No	No	Yes
SLAM-SL	Yes	No	Unclear	Yes	Yes
ODOL (OWL)	No	Yes	Yes	Yes	Yes
URN	No	No	Yes	Yes	Yes
LayOM	Yes	Yes	Yes	Yes	Yes
ADV	No	No	No	No	Yes

and SLAM-SL are based on the object-oriented language. Therefore, they are able to identify the objects of the design patterns distinctly.

6.1.2. Capability to Identify Structure Distinctly. This capability is present when the language description of the design pattern is sufficient to create a class diagram of the design pattern. Most of the languages based on mathematical formalism are not capable of identifying structure distinctly, the only exceptions being GEBNF and EOOPLG. GEBNF has a graphic component that makes it capable of identifying structures distinctly. All languages based on modeling languages support this capability. Modeling languages based on UML inherently provide class diagram support. Only three of the seven design pattern specification languages in the third group have this capability. The purpose of SPINE is to verify the design pattern through the behavior aspect; therefore, it does not have the capability to identify the structures of design patterns distinctly. SLAM-SL involves algebraic expressions for the formalization of design patterns. URN is based on GRL and UCM, which lacks the capability to identify structures distinctly. ADV provides individual object-level identification but cannot group these objects in a structure; rather, it focuses on interactions and collaborations at the object level.

6.1.3. Capability to Identify Collaborations Distinctly. The capability to identify collaborations distinctly refers to behavioral aspects of the design pattern. Method calls and parameters can help in identifying some collaborations; however, a simple way to know if the design-pattern specification language is capable of identifying all collaborations

Table X. Evaluation of the Core Properties of Design Pattern Specification Languages—Part B

	Comprehensiveness				Expressiveness		
Design Pattern	**Ability to handle multiple entities**	**Ability to address OO paradigm**	**Capability to formalize OO patterns**	**Other features**	**Graphical notation**	**Support Views**	**Textual Notation**
Languages Based on Mathematical Formalism							
LePUS	Yes	Yes	Yes	Yes	No	Yes	Yes
eLePUS	Yes	Yes	Yes	Yes	Yes	Yes	Yes
DisCo	No	Yes	No	No	No	Unclear	Yes
GEBNF	Yes	Yes	Yes	No	Yes	Yes	Yes
LOTOS	Yes	Yes	Yes	No	No	Yes	Yes
BPSL	Yes	Partial	Yes	No	No	Yes	Yes
Object-Calculus	Yes	Yes	Yes	No	No	Yes	Yes
EOOPLG	Yes	Yes	Unclear	No	No	Unclear	Yes
FSDP	Yes	Yes	Yes	No	No	No	Yes
Languages Based on Existing Modeling Languages							
DPML	Yes	Partial	Yes	Unclear	Yes	Yes	No
RBML	Yes	Yes	Yes	No	Yes	Yes	No
Constraint Diagram	Yes	Yes	Yes	No	Yes	Yes	No
DPDL	Yes	Yes	Yes	Yes	No	Yes	Yes
Languages Based on Other Languages							
RSL	Yes	Partial	Yes	No	No	No	Yes
SPINE	Yes	Yes	Yes	No	No	No	Yes
Slam-SL	Yes	Yes	Yes	No	No	No	Yes
ODOL (OWL)	Yes	Yes	Yes	No	No	Yes	Yes
URN	Yes	No	No	Yes	Yes	Partial	No
LayOM	Yes	Yes	Yes	Yes	No	Yes	Yes
ADV	Yes	Yes	Yes	Yes	No	Yes	Yes

distinctly is when it is able to produce a sequence diagram. Languages based on mathematical formalism mostly are capable of identifying collaborations distinctly; however, LOTOS, Object Calculus and FSDP do not have this capability. All languages based on existing modeling languages are capable of identifying collaborations distinctly. Most of the languages based on other languages are capable of identifying all collaborations between objects distinctly, except for SPINE, which can provide some collaborations through method calls and parameters [Shi 2007].

6.1.4. Original Formalism. The original formalism identifies whether the language introduces some original rules and specifications or if it is based on some existing rules and specifications and extends or modifies them to handle design patterns. Most languages based on mathematical formalism utilize original formalism. None of the existing modeling languages utilize original formalism, as they are based on modeling languages. All languages based on other languages utilize their own formalism, except for ADV and SPINE. ADV was developed to enhance design reuse by creating a specification for interfaces; SPINE extends the formalism defined in Prolog.

6.1.5. Textual Formalism. Textual formalism identifies if the language provides some textual formalism to describe a design pattern. All languages based on mathematical formalism utilize textual formalism because mathematical formalism is represented best in textual notations. Most of the languages based on modeling languages do not provide textual formalism. DPDL provides textual formalism, as it is based on XML,

Table XI. Comparison of Tool Support Features of Design-Pattern Specification Languages

	Ability to Create/ Generate DP	Ability to verify DP	Ability to detect DP	Ability to visualize DP	Ability to save DP	Ability to Open DP	Save Format	Tool Purpose	Tool Name
Languages Based on Mathematical Formalism									
LePUS	Not created by Author								
eLePUS	Not created by Author								
DisCo	Yes	Yes	No	Yes	No	No		Verification and code generation	DisCo Scanerio Tool
GEBNF	Not created by Author								
LOTOS	No	Yes	No	No	No	No	No	For checking behavioral consistency	LOTOS Simulator
BPSL	No	No	No	No	No	No	No	Generating Java code from BPSL specification	BPSL Tool
Object-Calculus	Not created by Author								
EOOPLG	Not created by Author								
FSDP	No	Yes	No	Yes	Yes	Yes	XML	For creating graphical notation & Java code	ANTLR
Languages Based on Existing Modeling Languages									
DPML	Yes	Yes	No	Yes	Yes	Yes	No	Design pattern modelling and instantiation	DPTool
RBML	Yes	Yes	Yes	Yes	Yes	No	No	Systematic use of patterns in UML model	RBML-PI and RBML-CC
Constraint Diagram	Yes	Yes	No	Yes	No	No	No	Creation and verification of design pattern	Constraint Diagram Tool
DPDL	No	No	No	Yes	No	Yes	DPDL (xml)	For creating UML class diagram & sequence diagram	DPDL Tool
Languages Based on Other Languages									
RSL	Not created by Author								
SPINE	No	Yes	No	No	No	No	NA	For verifying the pattern	HEDGEHOG
SLAM-SL	Yes	Yes	No	Yes	No	No	NA	To generate code, check conditions for incorporating design patterns in the project	SLAM-SL
ODOL (OWL)	Yes	Yes	Yes	Yes	Yes	Yes	RDF	To facilitate the use of pattern as knowledge artifacts	Online Repository
URN	Yes	Yes	No	Yes	No	No	No	To elicit, analyze, specify and validate the design pattern	jUCMNav
LayOM	Yes	No	No	No	No	No	No	To translate the design into C++ code.	
ADV	Yes	No	No	Yes	No	No	NA	To translate the design into C++ code.	ADV Tools

which was designed to describe data; however, RBML, DPML, and Constraint Diagram do not provide textual formalism. All languages based on other languages utilize textual formalism.

6.1.6. Ability to Handle Multiple Entities. Design patterns have the ability to handle multiple objects if they are able to group them into some form of set. All the languages in all groups have this capability. Languages based on mathematical formalism provide

this capability through sets. Languages based on UML usually add an extra layer to provide the capability of grouping the objects together.

6.1.7. Ability to Address the OO Paradigm. Some design patterns are based on OO concepts. If these concepts are not covered in the design-pattern specification language, the design pattern cannot be expressed completely and concisely. Examples of such design patterns are abstract factory, builder, and factory method. Most of the design patterns use certain concepts of object-oriented programming; therefore, a design-pattern specification language with this ability will be able to handle design patterns using the object-oriented paradigm. All design-pattern specification languages in all groups have the capability of addressing the object-oriented paradigm except URN, as URN is based on capturing scenarios against different goals and finding the best one, based on trade-offs. RSL is based on the RAISE language, which was designed for industrial software engineering. Its capability to handle all the object-oriented features, such as polymorphism and encapsulation, is not clear.

6.1.8. Capability to Formalize OO Patterns. The capability to formalize object-oriented patterns does not only depend on the ability of the design pattern to address the object-oriented paradigm but also on other features related to collaborations and interactions in the design pattern. DisCo is one of the languages that lacks the capability to formalize object-oriented patterns, as it separates objects from functions; this violates the object-oriented design philosophy. URN is the other language from the group of languages based on existing languages that does not have the ability to address the object-oriented paradigm.

6.1.9. Other Features. Although most design-pattern specification languages do not have additional features, some provide features that are beyond the requirements of a complete and concise definition of a particular design pattern. URN identifies the side effects of design patterns and can provide the performance aspects of design patterns. Similarly, in the LayOM language, the states of the variables of design patterns are captured over time, and also identify the default state of the variable.

6.1.10. Graphical Notation. This property checks if the design-pattern description language provides some sort of graphical symbol to describe the design pattern. This is not restricted to UML diagrams. Most of the languages belonging to the group based on mathematical formalism do not have a graphical notation, the exception being e-LePUS, which is based on LePUS and has added graphical notations. GEBNF is the other language belonging to this group, which has a graphical notation. GEBNF extends the BNF notation with graphical notations. UML-based languages belonging to the group of languages based on existing modeling languages have graphical notations, and include DPML and RBML. Constrain Diagram is another language that has a graphical notation. DPDL is based on XML, although it can generate class diagrams and sequence diagrams, but it does not provide a graphical notation. From the third group, the languages based on other languages, only URN provides a graphical notation. All other languages do not have graphical notations, mainly because these languages are based on a textual language and they have not been extended to include graphical notations.

6.1.11. Support Views. A design pattern has multiple aspects: structural, behavioral, and functional. Some languages only handle one aspect of design patterns; other languages handle multiple aspects and handle them distinctly. Some languages do not provide a distinction between these aspects, and expect the user to separate them. Most of the languages belonging to the group of mathematical formalism and modeling

Table XII. Relationship of Intent With Complexity in Design-Pattern Specification Languages

Language	Intent	Complexity
DPDL	Description	Low
GEBNF	Analysis	Medium
FSDP	Detection	Medium
DPML	Description	Medium
RBML	Description	Medium
Constraint Diagram	Description	Medium
SLAM-SL	Description	Medium
ODOL (OWL)	Description	Medium
LayOM	Description	Medium
LePUS	Verification	High
eLePUS	Description	High
DisCo	Analysis	High
LOTOS	Verification	High
BPSL	Analysis	High
Object-Calculus	Verification	High
EOOPLG	Verification	High
RSL	Verification	High
SPINE	Detection	High
URN	Analysis	High
ADV	Description	High

languages provide support for separate views, the only exception being FSDP. FSDP concentrates on the structural aspect of design patterns and uses the class diagram, represented in a formal way based on textual notation. From the group of languages based on other languages, RSL, SLAM-SL and URN do not provide support for multiple distinct views. RSL and SPINE, being verification languages, rely on structural semantics to identify the design patterns in the source code. Slam-SL syntax is a sequence of algebraic expressions in a single description. Identifying structural, behavioral, and other aspects has to be done manually. URN's main focus is to evaluate certain goals against specific scenarios, so that it can provide output on multiple goals and scenarios. However, the structural, behavioral, and functional aspects in software engineering terminology cannot be produced by the language.

6.1.12. Textual Notation. All the languages belonging to the group of languages based on mathematical formalism use textual notation for the specification of design patterns. Of the group of languages based on existing modeling languages, RBML, DPML, and Constraint Diagram have no textual notation. They rely only on graphical notation. DPDL, which lacks graphical notation, relies on textual notation. Almost all design-pattern specification languages based on other languages have textual notation, the only exception being URN, which has graphical notation only.

6.2. Individual Trends

As discussed in Section 1.3, design-pattern specification languages based on intent can be divided into four categories: description, analysis, detection, and verification.

The first trend that is observed is that languages with the intent of verification always have high complexity, as shown in Table XII. One of the reasons for that is that verification languages not only need to define the design pattern, they also need to define the sequence of steps to identify the design pattern in the source code, which

Table XIII. Relationship of Intent With Complexity in Design-Pattern Specification Languages

Language	Intent	Graphical Support
DisCo	Analysis	No
BPSL	Analysis	No
eLePUS	Description	No
SLAM-SL	Description	No
LayOM	Description	No
SPINE	Detection	No
LePUS	Verification	No
LOTOS	Verification	No
Object-Calculus	Verification	No
EOOPLG	Verification	No
GEBNF	Analysis	Yes
URN	Analysis	Yes
DPML	Description	Yes
RBML	Description	Yes
Constraint Diagram	Description	Yes
DPDL	Description	Yes
ODOL (OWL)	Description	Yes
ADV	Description	Yes
FSDP	Description	Yes
RSL	Description	Yes

adds complexity in the design pattern. This observation is strengthened by noting that most of the languages with the intent of description only have medium or low complexity. The languages with the intent of analysis usually fall into the medium- and high-complexity categories.

From Table XIII, we observe that the languages with the verification intent mostly lack graphical support. There is only one language with detection intent that does not have graphical support. This might be because the language objective is to identify design patterns instead of representing or describing them. Description languages fall into both categories, that is, sometimes they have graphical support and sometimes they lack graphical support, depending on the syntax on which they are based and the type of description that they are targeting.

Another trend, identified in Table XIV, is that although detection languages detect design patterns in languages, they do not provide information regarding collaboration between different objects distinctly. Languages with the purpose of description always provide information on collaboration in the design pattern distinctly.

One last individual trend that is observed is the capability of the languages to identify the structure individually. In Table XV, we observe that the languages with the analysis intent are always capable of identifying participants distinctly. One of the reasons for this could be that they analyze the structural aspects of the design pattern. This observation is further strengthened by noting that the only language with the analysis intent that does not have the capability of identifying the participants distinctly is URN. URN focuses on the analysis of design patterns against goals that do not need the structural aspects of the design pattern.

6.3. Collective Trend

We first try to determine if the notation of the design pattern specification language has an impact on the number of properties that are handled. From Table XVI, we see that

Table XIV. Relationship of Intent With the Capability to Identify Collaborations Distinctly in Design-Pattern Specification Languages

Language	Intent	Capability to identify collaborations distinctly
LOTOS	Verification	No
Object-Calculus	Verification	No
FSDP	Detection	No
SPINE	Detection	No
ADV	Description	No
SLAM-SL	Description	Unclear
LePUS	Verification	Yes
eLePUS	Description	Yes
DisCo	Analysis	Yes
GEBNF	Analysis	Yes
BPSL	Analysis	Yes
EOOPLG	Verification	Yes
DPML	Description	Yes
RBML	Description	Yes
Constraint Diagram	Description	Yes
DPDL	Description	Yes
RSL	Verification	Yes
ODOL (OWL)	Description	Yes
URN	Analysis	Yes
LayOM	Description	Yes

Table XV. Relationship of Intent With the Capability to Identify Participants Distinctly in Design-Pattern Specification Languages

Language	Intent	Capability to identify participants distinctly
LOTOS	Verification	No
Object-Calculus	Verification	No
FSDP	Description	No
DPML	Description	No
RBML	Description	No
SPINE	Detection	No
ODOL (OWL)	Description	No
URN	Analysis	No
ADV	Description	No
RSL	Verification	Partial
LePUS	Verification	Yes
eLePUS	Description	Yes
DisCo	Analysis	Yes
GEBNF	Analysis	Yes
BPSL	Analysis	Yes
EOOPLG	Verification	Yes
Constraint Diagram	Description	Yes
DPDL	Description	Yes
SLAM-SL	Description	Yes
LayOM	Description	Yes

Table XVI. Summary of Design-Pattern Specification Languages Against the Evaluation Framework with the Notation of the Language

Language	Number of framework properties handled	Language notation
LayOM	11	Textual
LePUS	10	Textual
eLePUS	10	Amphibious
GEBNF	10	Amphibious
DPDL	10	Textual
ODOL (OWL)	9	Textual
LOTOS	8	Textual
BPSL	8	Textual
Constraint Diagram	8	Graphical
Object-Calculus	7	Textual
EOOPLG	7	Textual
DPML	7	Graphical
RBML	7	Graphical
RSL	7	Textual
SLAM-SL	7	Textual
DisCo	6	Textual
FSDP	6	Textual
URN	6	Graphical
ADV	6	Textual
SPINE	5	Textual

the languages that have textual notation handle more properties than the languages that only have graphical notation.

One thing that should be kept in mind is that the intent of design-pattern specification languages is based on the purpose described by the authors of these languages; thus, we are not verifying the intent. For example, two languages with detection intent do not mean that they will have the same capability of detecting design patterns. It can be very possible that one language detects a few design patterns from the OO paradigm and another detects all the design patterns in the OO paradigm [Fulop et al. 2008]. Also, one design-pattern specification language may work on only Java code and the other works on UML diagrams only.

Second, we compare how many properties of the design-pattern evaluation framework are handled by each language. The first observation from Table XVII is that the complexity of design languages has nothing to do with the number of design-pattern evaluation framework features a design language handles. The second observation is that the languages with description intent usually handle more evaluation properties, which tells us that the evaluation framework features are geared more toward the description of the design pattern than their capability of detecting or verifying design patterns.

Finally, we compare how many times each property of the design-pattern evaluation framework is satisfied.

Table XVIII shows that design-pattern specification languages have the greatest ability to handle multiple entities. One interesting observation is that the capability to identify collaboration distinctly is satisfied 14 times, whereas identifying participants and structure distinctly is satisfied 10 times. One reason for this could be that identifying collaboration is more important in the detection, verification, and description of design patterns. Similarly, we see that textual notation and textual formalism are higher than graphical notation, as observed earlier.

Table XVII. Summary of Design-Pattern Specification Languages Against the Evaluation Framework

Language	Intent	Complexity	Number of framework properties handled
LayOM	Description	Medium	11
LePUS	Verification	High	10
eLePUS	Description	High	10
GEBNF	Analysis	Medium	10
DPDL	Description	Low	10
ODOL (OWL)	Description	Medium	9
LOTOS	Verification	High	8
BPSL	Analysis	High	8
Constraint Diagram	Description	Medium	8
Object-Calculus	Verification	High	7
EOOPLG	Verification	High	7
DPML	Description	Medium	7
RBML	Description	Medium	7
RSL	Verification	High	7
Slam-SL	Description	Medium	7
DisCo	Analysis	High	6
FSDP	Description	Medium	6
URN	Analysis	High	6
ADV	Description	High	6
SPINE	Detection	High	5

Table XVIII. Number of Times Design-Pattern Evaluation Framework Property is Satisfied

Property	Number of times design language satisfies the property
Ability to handle multiple entities	19
Textual formalism	18
Capability to formalize OO patterns	17
Textual notation	16
Ability to address OO paradigm	15
Capability to identify collaborations distinctly	14
Support views	13
Original formalism	11
Capability to identify participants distinctly	10
Capability to identify structure distinctly	10
Other features	6
Graphical notation	6

7. CONCLUSION

Design patterns are the interaction of classes and objects based on certain relationships. Therefore, in the realization of a design pattern, the actor performing the action is more than the implementation of the algorithm. This means that the implementation is important in the design pattern under the bounds of the protocol and sequences [Sterritt et al. 2010]. If we have to simplify the most important goals for software design-pattern description, these are as follows: generality, precision, and understandability.

Generality: Natural language is used in describing the traditional patterns along with graphical notations. The problem is that the important details of design patterns are hidden in the instantiation of objects; therefore, the pattern description should be generic to handle different usage scenarios of design patterns, but can also cover the specific details of instantiation. UML [UML 2003]–based object notations normally have limitations in describing design patterns abstractly. A pattern description should

be generic enough so that different applications and domain-specific realizations of the pattern can be achieved. The most common example is that the number of classes of concrete factory in abstract factory design pattern are not easy to handle using UML-based design-pattern specification languages [Erich Gamma 1994].

Precision: Precision is also important in the description of design patterns. When used for model transformation, the precise description of patterns is needed to derive the set of rules. Conventional object-oriented languages have many limitations in this area, such as the relationship between create product operation in the concrete factory and create product in concrete product class of abstract factory design pattern. However, association and dependencies are usually used in representing these relations in pattern description, which is not adequate. In short, the precise specification of pattern structure and behavior produces correct and accurate implementation of the design pattern.

Understandability: Although it can be argued that generality and precision are the most important goals for a pattern-description language, if the pattern is not understandable to developers and users, then the precision and generality is of no use. Therefore, we have added this as one of the goals of pattern-description languages. An unnecessarily complex design-pattern specification language will reduce the use of that language for pattern description and can be the reason for the incorrect use of the design pattern.

The computing discipline is changing rapidly and expanding into new territories such as mobile, cloud, distributed, and ubiquitous computing. This is giving rise to new problems and new design patterns to resolve these issues [Ali et al. 2014; Atkinson et al. 2015; Crane et al. 1995; Fehling et al. 2014; Meseguer 2014; Ravindran et al. 2014; Zaki and Forbrig 2011]. Therefore, there is a need for new design-pattern specification languages to handle emerging design patterns. This provides a reason for ongoing analysis of design-pattern specification languages and their categorization. This also raises the need for a unified framework for the evaluation of design patterns across different domains.

ACKNOWLEDGMENT

The authors would like to acknowledge the support provided by the Deanship of Scientific Research at King Fahd University of Petroleum and Minerals, Saudi Arabia.

REFERENCES

P. S. C. Alencar, D. D. Cowan, D. German, K. J. Lichtner, C. Lucena, and L. A. Nova. 1995. *Formal Approach to Design Pattern Definition & Application.* World Congress on Formal Methods. 576–594.

C. Alexander. 1977. *A Pattern Language*. Oxford University Press, New York, NY.

C. Alexander. 1979. *The Timeless Way of Building*. Oxford University Press, New York, NY.

M. M. Ali, A. F. Elsharkawi, M. El Said, and M. Zaki. 2014. Design patterns for multimedia mobile applications. *Journal of Computer Science* 1, 2.

H. Angel and J. J. Moreno-Navarro. 2007. Modeling and reasoning about design patterns in slam-sl. *Design Pattern Formalization Techniques* 206–235.

M. Aoyama. 2000. *Evolutionary Patterns of Design and Design Patterns.* IEEE, 110–116.

B. Appleton. 2000. *Patterns and Software: Essential Concepts and Terminology.* Accessed April 2016, http://www.sci.brooklyn.cuny.edu/~sklar/teaching/s08/cis20.2/papers/appleton-patterns-intro.pdf.

C. Atkinson, P. Bostan, and D. Draheim. 2015. Foundational MDA patterns for service-oriented computing. *Journal of Object Technology* 14, 1.

A. L. Baroni, Y. G. Gueheneuc, and H. Albin-Amiot. 2003. Design patterns formalization. *Rapport De Recherche, Département D'informatique, École Des Mines De Nantes*, 03, 03.

I. Bayley and H. Zhu. 2007. *Formalising Design Patterns in Predicate Logic.* IEEE, 25–36.

K. Beck and W. Cunningham. 1987. Using pattern languages for object oriented programs. *OOPSLA - Conference on Object-Oriented Programming, Systems, Languages, and Applications*.

M. L. Bernardi, M. Cimitile, and G. Di Lucca. 2014. Design pattern detection using a DSL-driven graph matching approach. *Journal of Software: Evolution and Process* 26, 12, 1233–1266.

A. Blewitt. 2007. *SPINE: Language for Pattern Verification*, IGI Global, Hershey, PA.

D. Bohdanowicz. 2005. *Toward Tool Support for Usage of Object-Oriented Design Patterns Expressed in Unified Modeling Language*, Blekinge Institute of Technology, Ronneby, Sweden.

J. Bosch. 1996a. *Language Support for Design Patterns*. (1996), 197–210.

J. Bosch. 1996b. Relations as object model components. *Journal of Programming Languages* 4, 39–61.

F. Buschmann, K. Henney, and D. C. Schmidt. 2007. *Pattern Oriented Software Architecture, Volume 5: On Patterns and Pattern Languages*, Wiley, Hoboken, NNJ.

A. Chihada, S. Jalili, S. M. H. Hasheminejad, and M. H. Zangooei. 2015. Source code and design conformance, design pattern detection from source code by classification approach. *Applied Soft Computing* 26, 357–367.

O. Coplien. 1992. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, New York, NY.

J. O. Coplien. 2002. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, New York, NY.

S. Crane, J. Magee, and N. Pryce. 1995. *Design Patterns for Binding in Distributed Systems*. Citeseer.

S. Deya and S. Bhattacharyab. 2010. Formal specification of structural and behavioral aspects of design patterns. *Journal of Object Technology* 9, 6, 99–126.

T. Di Noia, M. Mongiello, and E. Di Sciascio. 2014. *Ontology-Driven Pattern Selection and Matching in Software Design*, Springer.

J. Dietrich and C. Elgar. 2007. *An Ontology Based Representation of Software Design Patterns*, IGI Global, Hershey, PA.

A. H. Eden. 1999. *Precise Specification of Design Patterns and Tool Support in their Application*. Tel Aviv University, Tel Aviv.

A. H. Eden, J. Y. Gil, and A. Yehudai. 1996. *A Formal Language for Design Patterns*. Washington University, St. Louis, MO.

A. H. Eden, A. Yehudai, and J. Gil. 1997. *Precise Specification and Automatic Application of Design Patterns*. IEEE, 143–152.

R. H. Erich Gamma, Ralph Johnson, and John Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, New York, NY.

A. Flores, A. Cechich, and G. Aranda. 2007. *A Generic Model of Object-Oriented Patterns Specified in RSL*, IGI Global, Hershey, PA.

L. J. Fulop, R. Ferenc, and T. Gyimothy. 2008. *Towards a Benchmark for Evaluating Design Pattern Miner Tools*. IEEE, 143–152.

R. Gabriel. 1996. *Patterns of Software: Tales from the Software Community*. Oxford University Press, New York, NY.

E. Gasparis. 2007. *LePUS: A Formal Language for Modeling Design Patterns*, IGI Global, Hershey, PA.

A. H. Hannousse and Z. Liu. 2007. *Towards A Calculus for Design Patterns*. International Institute for Software Technology.

G. Hedin. 1997. Attribute extensions—a technique for enforcing programming conventions. *Nordic Journal of Computing* 4, 1, 93–122.

G. Hedin. 1998. Language support for design patterns using attribute extension. *Object-Oriented Technologies* 137–140.

S. Henninger and V. Corrêa. 2007. Software pattern communities: Current practices and challenges. *14th Conference on Pattern Languages of Programs (PLoP'07)*.

A. Herranz and J. J. Moreno-Navarro. 2003. Formal agility. how much of each. *Taller De Metodologias Agiles en el Desarrollo Del Software. JISBD* 47–51.

A. Herranz, J. Moreno, and N. Maya. 2002. Declarative reflection and its application as a pattern language. *Electronic Notes in Theoretical Computer Science* 76, 0, 197–215.

U. Itu-T. 2002. Focus group: Draft rec. Z. 151–goal-oriented requirements language (GRL). Geneva, Switzerland.

F. Jaafar, Y. G. Guéhéneuc, S. Hamel, F. Khomh, and M. Zulkernine. 2015. Evaluating the impact of design pattern and anti-pattern dependencies on changes and faults. *Empirical Software Engineering* 1–36.

D. Jing, Y. Sheng, and Z. Kang. 2007. Visualizing design patterns in their applications and compositions. *IEEE Transactions on Software Engineering* 33, 7, 433–453.

S. Kent. 1997. *Constraint Diagrams: Visualizing Invariants in Object-Oriented Models.* ACM, 327–341.

S. Khwaja and M. Alshayeb. 2013a. A framework for evaluating software design pattern specification languages. In *The 12th IEEE/ACIS International Conference on Computer and Information Science*. 16–20.

S. Khwaja and M. Alshayeb. 2013b. Towards design pattern definition language. *Software: Practice and Experience* 43, 7, 747–757.

D. K. Kim. 2007. *Role-Based Metamodeling Language for Specifying Design Patterns*, IGI Global, Hershey, PA.

D. K. Kim, R. France, S. Ghosh, and E. Song. 2003. A UML-based metamodeling language to specify design patterns. *Workshop Software Model Engineering (WiSME) with Unified Modeling Language Conference.*

S. K. Kim and D. Carrington. 2009. A formalism to describe design patterns based on role concepts. *Formal Aspects of Computing* 21, 5, 397–420.

S. Kodituwakku and P. Bertok. 2009. A mathematical approach to object oriented design patterns. *Journal of the National Science Foundation of Sri Lanka* 36, 3.

P. Kumar and A. Kumar. 2010. MCDMfJ : Mining creational design motifs from java source code. *International Journal of Computer and Network Security* 2, 1 (2010), 11–14.

L. Lamport. 1994. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems* 16, 872–923.

K. Lano. 2007. *Formalising Design Patterns as Model Transformations*, IGI Global, Hershey, PA.

A. Lauder and S. Kent. 1998. Precise visual specification of design patterns. *ECOOP'98—Object-Oriented Programming*, 114–134.

D. Lea. 1994. Christopher Alexander, an introduction for object-oriented designers. *Software Engineering Notes* 19, 1 (1994), 39–46.

C. Fehling, F. Leymann, R. Retter, W. Schupeck, and P. Arbitter. 2014. *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer.

J. K. Mak, C. S. Choy, and D. P. Lun. 2004. *Precise Modeling of Design Patterns in UML.* IEEE, 252–261.

D. Mapelsden, J. Hosking, and J. Grundy. 2002. *Design Pattern Modelling and Instantiation Using DPML.* Australian Computer Society, Inc., Sydney, Australia.

B. D. Martino and A. Esposito. 2015. A rule-based procedure for automatic recognition of design patterns in UML diagrams. *Software: Practice and Experience* 46, 7, 983–1007.

A. Mattsson, B. Lundell, B. Lings, and B. Fitzgerald. 2009. Linking model-driven development and software architecture: A case study. *IEEE Transactions on Software Engineering*, 35, 1, 83–93.

J. Meseguer. 2014. Taming distributed system complexity through formal patterns. *Science of Computer Programming* 83, 3–34.

S. J. Metsker. 2002. *Design Patterns Java Workbook*, Addison Wesley, New York, NY.

T. Mikkonen. 1998. Formalizing design patterns. In *Proceedings of the 20th International Conference on Software Engineering*. Los Alamitos, CA, USA.

J. L. I. Montes and F. L. G. Vela. 2003. Structural modeling of design patterns: REP diagrams. In *The 3rd Latin American Conference on Pattern Languages of Programming*.

OMG. 2005. *Object Constraint Language Specification, Version 2.0*. Available: http://www.omg.org/spec/OCL/.

Pattern Languages of Programs Conference. 1994. Monticello, Illinois, USA.

Y. Pan and E. Stolterman. 2013. *Pattern Language and HCI: Expectations and Experiences*, ACM, Paris, France.

N. Pettersson, W. Löwe, and J. Nivre. 2010. Evaluation of accuracy in design pattern occurrence detection. *IEEE Transactions on Software Engineering*, 36, 4, 575–590.

L. Prechelt, B. Unger, W. F. Tichy, P. Brossler, and L. G. Votta. 2001. A controlled experiment in maintenance: Comparing design patterns to simpler solutions. *IEEE Transactions on Software Engineering*, 27, 12, 1134–1144.

R. R. Raje and S. Chinnasamy. 2001. *eLePUS—a Language for Specification of Software Design Patterns*, ACM, Las Vegas, NV.

G. Rasool and D. Streitferdt. 2011. A survey on design pattern recovery techniques. *Journal of Computer Science* 8, 2.

R. Ravindran, R. Suchdev, Y. Tanna, and S. Swamy. 2014. *Context Aware and Pattern Oriented Machine Learning Framework (CAPOMF) for Android.* IEEE, 1–7.

D. Riehle. 1997. *Composite Design Patterns.* ACM, 218–228.

D. Riehle and H. Zullighoven. 1996. Understanding and using patterns in software development. *Theory and Practice of Object Systems* 2, 1, 3–13.

J. F. Roy, J. Kealey, and D. Amyot. 2006. Towards integrated tool support for the user requirements notation. *System Analysis and Modeling: Language Profiles* 198–215.

M. Saeki. 2000. *Behavioral Specification of GOF Design Patterns with LOTOS*, IEEE Computer Society, Washington, DC.

M. Saeki, T. Hiroi, and T. Ugai. 1993. *Reflective Specification: Applying a Reflective Language to Formal Specification.* IEEE, 204–213.

A. Seffah and M. Taleb. 2012. Tracing the evolution of HCI patterns as an interaction design tool. *Innovations in Systems and Software Engineering* 8, 2, 93–109.

N. Shi. 2007. *Reverse Engineering of Design Patterns from Java Source Code*. University of California, Davis.

A. V. Sriharsha and A. R. M. Reddy. 2015. Empirical analysis of design pattern metrics for building modest formalized catalog. *International Journal of Information Research and Review* 2, 10, 1232–1236.

A. Sterritt, S. Clarke, and V. Cahill. 2010. Precise specification of design pattern structure and behaviour. *Modelling Foundations and Applications* 277–292.

T. Taibi. 2007. *An Integrated Approach to Design Patterns Formalization*, IGI Global, Hershey, PA.

T. Taibi and D. C. Ngo. 2003. Formal specification of design pattern combination using BPSL. *Information and Software Technology* 45, 3, 157–170.

OMG UML. 2003. Superstructure Specification Version 2.4.1. Available: http://www.omg.org/spec/UML/2.4.1/2011.

W3C. 2008. *Extensible Markup Language (XML) Version 1.0*. Available: http://www.w3.org/TR/REC-xml/2008.

W3C. 2004. OWL Web Ontology Language Guide. Available: https://www.w3.org/TR/owl-guide/.

W3C. 2005. Resource Description Framework (RDF). Available: https://www.w3.org/RDF/.

D. Yu, Y. Zhang, and Z. Chen. 2015. A comprehensive approach to the recovery of design pattern instances based on sub-patterns and method signatures. *Journal of Systems and Software* 103 1–16.

M. Zaki and P. Forbrig. 2011. *User-Oriented Accessibility Patterns for Smart Environments*, Springer.

H. Zhu and L. Shan. 2006. Well-formedness, consistency and completeness of graphic models. *Proceedings of UKSIM* 6, 47–53.

Received December 2015; revised March 2016; accepted April 2016