J.Natn.Sci.Foundation Sri Lanka 2008 36 (3):219-227

RESEARCH ARTICLE

A mathematical approach to object oriented design patterns

Saluka R. Kodituwakku[1]* and Peter Bertok[2]

[1] *Department of Statistics and Computer Science, Faculty of Science, University of Peradeniya, Peradeniya.*

[2] *Department of Computer Science, RMIT University, Melbourne, Australia.*

Revised: 27 May 2008 ; Accepted: 18 July 2008

Abstract: Although design patterns are reusable design elements, existing pattern descriptions focus on specific solutions that are not easily reusable in new designs. This paper introduces a new pattern description method for object oriented design patterns. The description method aims at providing a more general description of patterns so that patterns can be readily reusable. This method also helps a programmer to analyze, compare patterns, and detect patterns from existing software programmes.

This method differs from the existing pattern description methods in that it captures both static and dynamic properties of patterns. It describes them in terms of mathematical entities rather than natural language narratives, incomplete graphical notations or programme fragments. It also helps users to understand the patterns and relationships between them; and select appropriate patterns to the problem at hand. We also present a case study to demonstrate the methods' suitability for specifying object oriented design patterns.

Keywords: Class diagrams, design pattern, pattern formalism

INTRODUCTION

Design patterns[1-7] are in essence design forms for describing successful designs. The advantage of using pattern forms to document good designs is that they can be more easily shared and widely used by software developers. In the Design Patterns book[1], the description of patterns uses a standard form that consists of the following sections: Pattern Name and Classification, Intent, Also Known As, Motivation, Applicability, Structure, Participants, Collaborations, Consequences, Implementation, Sample Code, Known Uses, and Related Patterns. Using this form, a design pattern names, abstracts and identifies the key aspects of a common design structure; it identifies the participating classes and instances, their roles and collaborations, and the distribution of responsibilities. For ease of learning and reference, in[1], design patterns are organized into a catalogue.

The catalogue is formed by two criteria: purpose and scope. According to their purpose, design patterns are classified into creational, structural and behavioural, whereas the scope divides patterns into class patterns and object patterns. Class patterns capture relationships between classes and their subclasses; object patterns deal with relationships between objects.

Software patterns provide reusable solutions to recurring design problems. Even if we have reusable solutions they may not be effectively reused, because it is difficult to find an existing solution to a problem. Pattern catalogues[1,5,8] group patterns based on their intent or scope rather than based on their relationships. Such collections can assist the developer to find individual patterns, but they are almost useless for selecting a collection of patterns to fit a complex design problem. Pattern languages (PLs)[9-12] organize patterns according to the relationships between patterns. Hence, PLs can be applied to solve complex design problems. However, most of the identified and documented patterns have been documented as individual patterns. While numerous patterns remain standalone and not achieving their full application potential, researchers put more emphasis on documenting new patterns and pattern languages, than on organizing these patterns into languages or other efficient collections.

When selecting a collection of related patterns to solve a complex problem, relationships between patterns must be understood before applying them. Conventional pattern descriptions do not reveal easily how patterns relate to each other. Although there are standard methods

*Corresponding author

for documenting object oriented designs, pattern authors do not use them in their full potential to specify patterns. For instance, Unified Modeling Language (UML)[13-15] provides a collection of modeling languages that facilitate specifying static and dynamic properties of object oriented designs. However most of the traditional patterns are described by means of class diagrams, program fragments and natural language narratives. State diagrams, collaboration diagrams and interaction diagrams are not used by many of the pattern authors. Therefore, the traditional descriptions are informal and sometimes difficult to understand. As a result users encounter problems in understanding patterns as well as the relationships among patterns. Even though the UML could successfully be used to provide standard description of object oriented design patterns researchers have attempted to provide formal methods rather than using UML like standards for pattern descriptions.

Pattern formalisms have been seen as a better solution, but the proposed approaches[16-25] still fail because less emphasis has been put on understanding and identification of relationships between patterns. In order to assist developers to find suitable patterns quickly and easily, it is necessary to introduce new pattern descriptions or new patterns classification and organization approaches that are not subject to these limitations.

Since the publication of the Design Patterns book[1], some research effort has been made to find a better description and organization of design patterns and to facilitate a more effective use of patterns. Some researchers have found Gang of Four (GOF's) pattern description informal and ambiguous and have proposed various formal descriptions. However, these formalisms fail to address the problem successfully. In this paper, we propose an approach based on mathematical concepts by extending the relations proposed in[19]. This extended set of mathematical relations facilitates the description of all structural and behavioural components of patterns. Analyzing and comparing these relations can also identify relationships such as Uses, Refines and Variants.

The rest of the paper is organized as follows. Section 2 describes existing formal approaches, their advantages and disadvantages. Section 3 summarizes the basic abstractions related to object oriented design. It also explains the mathematical concepts used and the relationships between those concepts. Section 4 shows how design abstractions are combined to form patterns, and formal description of design patterns. Section 5 presents a case study that demonstrates the applicability of the proposed methods. Section 6 presents the conclusions.

An approach for formalizing patterns based on the temporal logic of actions has been proposed[22]. In this approach, each pattern is formalized as units of modularity. A pattern is formalized as a behavioral layer introducing slices of objects that resemble program slices. In this specification, three concepts, specified as Classes, Relations and Actions, are used to specify patterns. Classes and Actions represent the objects and methods involved in the solution structure. Relations are directly derived from the actions. For instance, a method "request" results in an action Request and a relation Requested.

A formal approach to architectural patterns is proposed in[16]. This approach is based on an object-oriented model integrated with a process-oriented method for describing patterns. The proposed approach comprises Abstract Data Views and Abstract Data Objects (ADV/ ADO), specified in specialized scheme language. ADVs and ADOs are substituted into design patterns realization while maintaining a separation between view and object.

A formal specification language for object oriented design pattern is discussed in[18-21,23,24]. In this language (LePUS), higher order logics are used to specify patterns as mathematical formulae. The fundamental design elements such as classes, methods and inheritance hierarchies, are specified as sets and functional relations between them. Fundamental design elements and relationships between them are specified as predicates. These predicates are combined to specify patterns as mathematical formulae. Each formula was accompanied with a specified visual notation by illustrating the structure of the pattern.

In the classic object model[17] is extended with the concepts of states and layers to support the specification of patterns. A state in this approach is an abstraction of the internal state of the object. The layers encapsulate the objects and intercept messages. The layers are used to represent relationship between objects and are classified into structural relations, behavioural relations and application domain relations. Structural relation types define the structure of a class. Behavioural relation types are used to relate an object to its clients. Behavioural relations restrict the behavior of the class. Application domain relations are used to specify application domain classes.

Almost all the proposed formal approaches specify the relations between objects and methods, but it does not address the relations among methods. LePUS formulae[18] address most of static and dynamic properties of design patterns. However, complex mathematical expressions make it difficult to understand. This specification is not sufficient for describing some restrictions. For instance,

this approach facilitates the specification of method invocations, but does not enable the description of restricted method invocation. Furthermore, mathematical relations used in this specification are not sufficient for detecting relationships such as Variants and May-Use.

The ADV/ADO approach[16] describes fundamental design constructs as algorithms. Since such descriptions are difficult to compare, this method does not help users to identify relationships between design patterns. In our description, patterns' participants and their collaborations are specified in terms of mathematical entities. Since mathematical entities can easily be analyzed and compared, the proposed method assists users to compare properties of patterns and hence to identify relationships between patterns.

The expressiveness of [17] specification language is demonstrated by phrasing constraints of the pertaining objects. His work restricts rules to the scope of a single object at a time, thereby serving only a limited number of cases where no higher-level perspective is incorporated in the design patterns. In addition, this approach primarily focuses on the implementation of design patterns rather than formal representation of patterns.

METHODS AND MATERIALS

The use of standard modeling methods such as UML could be used to describe the static and dynamic properties of object oriented designs. Even if such descriptions are complete there is no uniformity. In other words, static properties and dynamic properties could be described by means of different modeling languages. For instance class diagrams and interaction diagrams are completely different. So we attempted to describe the solution section of object oriented design patterns in uniform manner. In order to provide unambiguous and uniform description, we describe the structure of object oriented design patterns in terms of mathematical entities: sets and relations. The set of relations used in describing patterns is an extension of the relations proposed in[18].

Pattern elements and description primitives: The solution structure of an object oriented design pattern can be abstracted as follows. The pattern's participants are entities such as classes, functions or methods and attributes, and collections of them. The pattern's collaborations describe how participants work together to solve the expected task. The structure of the solution describes the relationships (inheritance, aggregation etc.) between the participants.

Participants and collaborations: A pattern's participants are the fundamental design components on one hand and their responsibilities on the other. Usually, participants include classes, objects, methods and attributes. For example, participants of a pattern may contain a collection of classes and a collection of methods defined on those classes. In this way, a pattern's participants can be viewed as a collection of sets. These elements often have a particular property. For instance, a collection of classes derived from a particular abstract class and a collection of methods with identical signatures redefined in a set of classes is common between GOF[1] and other closely related patterns[5]. On the other hand, a uniform set is a collection of entities, called members that have a particular property. Thus, uniform sets enable the description of participants.

Additionally, higher order sets such as a collection of hierarchies whose objects are produced by a set of methods and a set of sets of methods redefined in a set of classes are also common among design patterns. In order to describe these collections of design entities, a set of first order and higher order uniform sets given in Table 1 are defined.

Associations between participants describe the organization of pattern's participants: their relationships and collaborations. Relationships describe how participants relate to each other and collaborations describe the way participants interact with each other. For example, relationships such as inheritance and aggregation are often used in pattern descriptions. Usually such relationships exist between two sets of classes or objects. Interactions between objects, between

Table 1: Sets in object oriented designs

Set	Description
P	the domain of participants; includes classes, methods and attributes
C	the domain of classes; it is a subset of P
M	the domain of methods; it is also a subset of P
H	the domain of class hierarchies
2^C	the domain of sets of classes
2^H	the domain of sets of class hierarchies
2^{2M}	the domain of sets of method families

methods, and between objects and methods explain how participants interact with each other to carry out the expected task.

Representation of participants: Since uniform sets facilitate the description of participants, mathematical relations enables the description of relationships and interactions exist among participants. It is also argued that relationships and interactions between participants can be specified in terms of mathematical relations [18]. Furthermore, class and method definitions can also be specified as relations. Unary relations facilitate the description of abstract and concrete classes. Since methods are defined in classes, they can be specified as binary relations. More precisely, definition of participants and their collaborations can be specified as a set of mathematical relations. This section describes how pattern's participants and their collaborations can be described as relations.

Inheritance relation: When two sets of classes relate to each other by inheritance, the relationship can be expressed as a binary relation between two sets of classes.

Definition

'*Inheritance*: A × B' indicates that class B inherits from class A.

Inheritance relation is described with the base class and subclass in the following way:
Inheritance (base-class-name, sub-class-name)

Note that the *Inheritance* relation is sufficient to specify both single and multiple inheritance. Multiple inheritance can be represented as a set of Inheritance relations.

Reference relation: The relationship between container class and contained objects can be expressed as a binary relation between two sets of classes.

Definition

'*ReferenceTo*: $C_r \times C_d$ indicates that class C_r maintains a reference to class C_d. In other words, class C_r defines a member whose type is a reference to an instance of class C_d.

The *Reference To* relation can be used to describe the relationship between a container class and a single contained class. When a single container class contains many contained classes, these reference relations can be expressed as a collection of *ReferenceTo* relations. To make our description more concise, we define a single relation to express these reference relations.

Definition

'*ReferenceToMany*: $C_r \times C_d$' indicates that class C_r maintains references to a set of classes C_d.

Reference To and *ReferenceToMany* relations are described with the container class and contained class in the following way:
ReferenceTo(container-class-name, contained-class-name) and *ReferenceToMany (container-class-name, a set of contained-class-names).*

Abstract class as a relation: An abstract class can be expressed as a unary relation.

Definition

'*Abstract*: *C*' indicates that the class is an abstract class.

The *Abstract* relation is described with the name of the abstract class in the following way:
Abstract (class-name)

Concrete class as a relation: A concrete class can be expressed as a unary relation.

Definition

'*ConcreteClass*: *C*' indicates that the class is a concrete class.

When a pattern has a non-abstract class which is not related to any other class by inheritance or reference relation, it is described as *ConcreteClass* relation with its name in the following way:

ConcreteClass (class-name)

Inheritance class hierarchy as a relation: Since an inheritance hierarchy consists of root class, an Abstract class, and a set of subclasses, it can be described as a composition of a set of *Inheritance* relations and an *Abstract* relation.
We combine all inheritance relations into a single relation. Whenever a set of classes forms a hierarchy it is specified as a composition of an *Abstract* or *ConcreteClass* relation and *Hierarchy* relations.

Definition

'*Hierarchy*: *H*' indicates that the set of classes forms an inheritance hierarchy.

Inheritance class hierarchy is described with its root and leaves in the following way:
Hierarchy (concrete-class-names),
Abstract (root-class-name)

Shared method as a relation: When an inheritance hierarchy, or a set of classes, defines a method with a signature and provides different implementations, this can be expressed as a binary relation between a set of methods and a set of classes.

Definition

'SharedMethod: $M \times H'$ indicates that method M is shared by the classes in hierarchy H.

A shared method is described with the method and the set of classes in the following way:
Shared-Method (method-name, class-names)

Method definitions as relations: When a method is defined in a concrete class, the relationship between the class and the method can be expressed as a binary relation.

Definition

MethodOf: $M \times C'$ indicates that method M is defined in class C.

MethodOf relation is described with the method name and the class name in the following way:
MethodOf (method-name, class-name)

Arguments of a method as a relation: When a method has a parameter that is an instance of another class, the relationship between the method and the argument object can be expressed as a relation. If the method takes more than one parameter, it is necessary to specify the order of parameters. The relationships in such cases are defined as relations between three sets: set of methods, set of classes and the set of natural numbers. The set of natural numbers is used to specify the order of arguments such as 1st, 2nd and so on.

Definition

'Argument: $N \times M \times C'$ indicates that method M takes an instance of class C as its n^{th} ($n \in N$) argument.

An *Argument* relation is described with the method, the argument-object and the position of the argument in the argument-list in the following way:
Argument (argument-number, method-name, argument-object), where argument-number indicates the position of the argument (first, second and so on).

MethodInvocation relation: When a method invokes another method, that interaction can be expressed as a binary relation between two sets of methods.

Definition

'MethodInvocation: $M_s \times M_t'$ indicates that method M_s invokes method M_t.

MethodInvocation relation is described with the calling-method and the called-method in the following way:
MethodInvocation (calling-method-name, called-method - name)

Forwarding relation: When a particular method calls another method-and passes its arguments to the called method; the Method Invocation relation does not address the relationship between arguments of the two methods. This type of method invocation has to be expressed as a separate binary relation.

Definition'

'Forwarding: $M_s \times M_t'$: indicates that method M_s invokes method M_t and the actual arguments in the invocation expression are the arguments defined for M_s.

The *Forwarding* relation is described with the calling-method and the called-method in the following way:
Forwarding (calling-method-name, called-method-name)

PreventInvocation relation: When a method is implemented to invoke another method or to prevent such an invocation, the *MethodInvocation* relation is not sufficient to describe that interaction. In order to describe such restrictions, we combine the *MethodInvocation* relation with the *PreventInvocation* relation. The *PreventInvocation* specifies that a method does not allow the invocation of another method.

Definition

'PreventInvocation: $C \times M_1 \times M_2'$ indicates that the method M_1 defined in class C prevent an invocation of method M_2.

The *PreventInvocation* relation is described with names of the methods and the class that contains the calling method in the following way:
PreventInvocation (class-name, calling-method-name, called-method-name)

Creation relation: Object creation can be expressed as a binary relation between a set of methods and a set of classes.

Definition

'Creation: $M \times C'$ indicates that method M creates instances of class C.
Creation relation is described with the method and type of the object to be created in the following way:
Creation (method-name, type-of-object)

Return type relation: When a method returns instances of a class as its return type, the relationship between

the method and the object returned can be expressed as a binary relation between a set of methods and a set of classes.

Definition

'Return-Type: M × C' indicates that method M returns an instance of class C.

Return-Type relation is described with the method and type of the returned object in the following way:
Return-Type (method-name, type-of-object)

Production relation: When a method creates and returns instances of a class, this can be expressed as a combination of the *Creation* and *Return Type* relations.

We combine the two relations into a single binary relation.

Definition

'Production: M × C' indicates that method M creates and returns an instance of class C.

Production relation is described with the method and type of the produced object in the following way:
Production (method-name, type-of-object)

Assignment relation: When a class maintains a reference to another class, one of its methods initializes or assigns value (object) to the reference variable.
The collaboration between two classes and the method can be expressed as a relation between a set of methods and two sets of classes.

Definition

'Assignment: M × C_r × C_d' indicates that method M defined in class C_r assigns an instance of class C_d to an instance variable that is a reference to class C_d.

Assignment relation is described with the two classes and the method in the following way:
Assignment (method-name, container-class, contained-class)

Note: If any of the binary relation is a one-to-one relation, it is specified as $R^{\wedge\{1\text{-}1\}(S\_1, S\_2)}$. In other words, $R^{\wedge\{1\text{-}1\}(S\_1, S\_2)}$ specifies that relation R between sets S_1 and S_2 is bijective.

A formal description of patterns: In the previous section, we described the pattern elements and descriptive primitives, and the corresponding mathematical relations. These fundamental building blocks and relations occur repeatedly across solution structures of object oriented design patterns. For example, a class may know of another class *via* an interface (abstract class) or a message received may be delegated to a component object for detailed processing. These fundamental building blocks are combined in various ways when documenting different design patterns. After explaining how design abstractions can be described as relations, the next step is to combine these relations into a simple and unambiguous formal description: patterns will be specified as composition of design abstractions.

In our formal description of patterns, each description stands for the complete and final specification of the solution section of a single design pattern. A formal description of this kind consists of some or all of the sets and relations discussed in the previous section. The description of a pattern has the following form:

Pattern-name (p_1, p_2, ..., p_k)
{
$R_1(p_1, p_2, ..., p_k)$,
$R_2(p_1, p_2, ..., p_k)$,
$R_3(p_1, p_2, ..., p_k)$,
.....
.....
$R_n(p_1, p_2, ..., p_k)$,
}

In the above specification, (p_1, p_2, ..., p_k represent a set of fundamental building blocks that describe the pattern's participants, i.e. hierarchies, classes, methods and so on. R_1, R_2,.....,R_n represent the relations and interactions between the participants described in page 229.

Case study: applying the methodology to GOF patterns

In order to illustrate the applicability of our methodology, it is applied to two patterns, Observer and Adapter, selected from the GOF catalogue. This section presents the description of the Observer and Adapter patterns. Due to the limited space, only the Observer pattern is discussed in detail.

Description of the observer pattern

Intent: The Observer pattern[1] defines a dependency between a subject and a number of observer objects such that whenever the subject changes state, all the observers are notified of the change and can take appropriate action.

A subject can have any number of observers. The subject maintains a reference to each of its observes so

it can send notifications to all its observers. Once an observer is notified about a change of state, it queries the subject for information and updates its state to become consistent with that of the subject. In order to query from the subject, each observer maintains a reference to its subject. Subjects and observers implement the notify/ update protocol to keep the state consistent.

Observer structure contains a hierarchy of observers and subjects. Each subject and its observers defines a collection of methods. These sets of classes and methods are specified as sets in the following way.

Observers $\in$ H; *Subject, ConcreteSubject* $\in$ C
attach, detach, notify, getState, setState, update $\in 2^M$
A collection of observers is designed as a hierarchy. The abstract class observer defines the common interface for updating the state. The ConcreteObservers implement the observer interface for querying and updating. All observers define a method update () with the same signature but may provide different implementations. This means that update () method is shared by all observers. The inheritance hierarchy and its shared methods can be described with (*Hierarchy, Abstract*) and *Shared-Method* relations.
Hierarchy(Observers); Abstract(Observer)
Shared-Method(update, Observers)
In order to observe its observers, each subject defines an interface for attaching and detaching observers. Additionally, the interface contains a method for change of state notification: whenever a subject changes its state it has to update its state and to provide information about the new state. Concrete Subject is derived from the subject and it extends the subject's interface to update its state and to answer queries from its observers. This means that all subjects share the methods in the subject's interface. Relationships between subjects and concrete subjects are described as an *Inheritance* relation. The shared interface can be specified as a collection of *Shared-Method* relations.

Inheritance(Subject, ConcreteSubject)
Shared-Method(attach, Subjects)
Shared-Method(detach, Subjects)
Shared-Method(notify, Subjects)
Shared-Method(getState, ConcreteSubjects)
Shared-Method(setState, ConcreteSubjects)

The subject maintains references to its observers by defining an instance variable, which is a reference to the observer. The *ReferenceToMany* relation enables the description of this relationship.

ReferenceToMany(Subject, Observers)
In order to query from its subject, each observer maintains a reference to its subject. This relationship can be described with the *ReferenceTo* relation.
ReferenceTo(Observers, ConcreteSubjects)
The setState () method updates the state and invokes the notify () method to inform its observers about the change of state. In turn the notify () method invokes the update () method of all attached observers. When an observer is notified about the change of state, the update () method invokes getState () to get the new state and uses it to reconcile its state with that of the subject. Interactions between these methods are explained as a set of *Method-Invocation* relations.
Method-Invocation(setState, notify)
Method-Invocation(notify, update)
Method-Invocation(update, getState)
The attach () and detach () methods defined in Subject take observers as the first argument. Similarly, the update () method defined in observer takes a subject as the first argument. The relationships between these methods and objects can be described as *Argument* relations.
Argument(1, attach, Observers)
Argument(1, detach, Observers)
Argument(1, update, Subject)
A subject attaches its observers by assigning observer instances to its reference variables. The interaction is explained as an *Assignment* relation.
Assignment(attach, Subject, Observers)
Finally, all these sets and relations can be combined to specify the Observer pattern in the following way.
Observer (Observers $\in$ *H, Subject, ConcreteSubject* $\in$ *C, attach, detach, notify* $\in 2^M$*, getState, setState, update* $\in 2^M$*)*
{
Hierarchy(Observers), Abstract(Observer)
Inheritance(Subject, ConcreteSubject)
Shared-Method(attach, Subjects)
Shared-Method(detach, Subjects)
Shared-Method(notify, Subjects)
Shared-Method(getState, ConcreteSubjects)
Shared-Method(update, Observers)
Shared-Method(setState, ConcreteSubjects)
ReferenceTo(Observers, ConcreteSubject)
ReferenceToMany(Subject, Observers)
Argument(1, attach, Observers)
Argument(1, detach, Observers)
Argument(1, update, Subject)
Method-Invocation(setState, notify)
Method-Invocation(notify, update)
Method-Invocation(update, getState)
Assignment(attach, Subject, Observers)
}

Description of the adapter pattern: The Adapter[1] pattern converts the interface of an existing class to another interface that is compatible with the interface of the clients.

This allows the cooperation of classes that otherwise could not do so, due to minor differences between the interface provided and the interface expected. The Adapter pattern provides two alternative solutions, Class Adapter and Object Adapter, for converting interfaces. The formal descriptions of these two solutions are given below.

Class Adapter*(Target, Adapter, Adaptee* $\in$ *C, requestInit, requestFor, specialRequest* $\in 2^M$*)*
{
ConcreteClass(Adapter), ConcreteClass(Adaptee),
Inheritance(Target, Adapter), Inheritance(Adaptee, Adapter),
MethodOf(specialRequest, Adaptee), MethodOf (requestInit, Target),
MethodOf(requestFor, Adapter), MethodInvocation (requestInit, requestFor),
MethodInvocation(requestFor, specialRequest)
}

Object Adapter*(Target, Adapter, Adaptee* $\in$ *C, requestInit, requestFor,*
specilRequest $\in 2^M$ *)*
{
ConcreteClass(Adapter), ConcreteClass(Adaptee),
Inheritance(Target, Adapter), ReferenceTo(Adapter, Adaptee),
MethodOf(specialRequest, Adaptee), MethodOf (requestInit, Target),
MethodOf(requestFor, Adapter), Method Invocation (requestInit, requestFor),
MethodInvocation(requestFor, specialRequest)
}

Explanation

Target, Adapter, Adaptee $\in$ *C* specifies that the pattern contains three classes. RequestInit, requestFor, specilRequest $\in 2^M$ specify that the pattern contains a set of sets of methods. The provided interface can be defined as an abstract class or a concrete class. However, the class that converts the interface and the class with the expected interface must be concrete classes. That is described by the *ConcreteClass (Adapter)* and *ConcreteClass (Adaptee)* relations. In case of a Class Adapter, a new class is inherited from the class with the provide interface and class with the expected interface. In other words, multiple inheritance is used to convert the interface of an existing class into the interface that its clients expect. The *Inheritance (Target, Adapter)* and *Inheritance (Adaptee, Adapter)* relations describe this formally. As an alternative, a new class is inherited from the class with the provide interface which maintains a reference to the class with expected interface. The *Inheritance (Target, Adapter)* and *ReferenceTo (Adapter, Adaptee)* describe the relationships between three classes. This is referred to as Object Adapter. Clients initialize a request with the provided interface, and this request is forwarded to the class with expected interface that carry out the requested task. Each of the three class interfaces can have any number of methods, and the MethodOf *(requestInit, Target), MethodOf (requestFor, Adapter)* and MethodOf *(specialRequest, Adaptee)* relations describe the methods defined in these classes. The methods communicate with each other to carry out the requested task. The *requestInit ()* method passes the request to the adapter class by invoking the *requestFor ()* method. In turn, the *requestFor ()* method invokes the *specialRequest ()* method. *MethodInvocation (requestInit, requestFor)* and *MethodInvocation (requestFor, specialRequest)*relations describe these interactions in our formal description.

CONCLUSION

In this research study we investigated fundamental design constructs in object-oriented designs. Then we investigated an appropriate descriptive form for object oriented design patterns that facilitates the description of static and dynamic properties in a uniform manner. We proposed a new description method for object oriented design patterns. The proposed method specifies patterns in terms of mathematical relations rather than object notations and textual descriptions. The methodology was applied to two well-known patterns and the resultant descriptions are also presented. From the investigation and application of the methodology the following conclusions were made:

- Static and dynamic properties of object oriented design patterns can successfully be specified in terms of mathematical relations.
- The proposed approach has significant advantages over the other proposed formal approaches as it captures participants and all kinds of interactions.
- The method can be used as the basis for developing tools support for patterns. For example, it can be used to detect pattern instances in software systems and to generate automatic codes for patterns.

References

1. Gamma E., Helm R., Johnson R. & Vlissides J.M. (1994). *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley, USA.
2. Alexander C. (1979). *The Timeless Way of Building*. Oxford University Press, UK.
3. Alexander C., Silverstein M. & Ishikawa S. (1977). *A Pattern Language*. Oxford University Press, UK.

4. Fowler M. (1997). *Analysis Patterns: Reusable Object Modules*. Addison-Wesley, USA.
5. Buschmann F., Meunier R., Rohnert R. & Sommerlad P. (1996). *A System of Patterns: Pattern Oriented Software Architecture*. Addison-Wesley, USA.
6. Pree W. (1994). *Design Patterns for Object Oriented Software Development*. Addison-Wesley, USA.
7. Rising L. (2000). *The Pattern Almanac*. Addison-Wesley, USA.
8. Riehle D. & Züllighoven H. (1996). Understanding and using patterns in software development. *Theory and Practice of Object Systems* **2**: 3-13.
9. Coplien J. O. & Schmidt D. C. (1995). *Pattern Languages of Program Design*. Addison-Wesley, USA.
10. Vlissides J. M., Coplien J. O. & Kerth N. L. (1996). *Pattern Languages of Program Design 2*. Addison-Wesley, USA.
11. Martin R., Riehle D. & Buschmann F. (1998). *Pattern Languages of Program Design 3*. Addison-Wesley, USA.
12. Harrison N., Foote B. & Rohnert H. (2000). *Pattern Languages of Program Design 4*. Addison-Wesley, USA.
13. Pilone D. & Pitman N. (2005). *UML 2.0 in a Nutshell,* first edition. O'Reilly Media, Inc, USA.
14. Ambler S.W. (2005). *The Elements of UML(TM) 2.0 Style*. Cambridge University Press, UK.
15. Martin Fowler (2003). UML Distilled: *A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Professional, USA.
16. Alencar P. S. C., Cowan D. D. & Lucena C. J. P. (1996). A Formal Approach to Architectural Design Patterns. *Proceedings of the 3rd International Symposium of Formal Methods*. pp. 576-594.
17. Bosch J. (1996). Language Support for Design Patterns, *Proceedings of the Technology of Object-Oriented Languages and Systems.*
18. Eden A.H., Gil J. & Yahudai A. (1997). A Formal Language for Design Patterns. *Technical report* WUCS-97-07, Washington University, St. Louis, Missouri, USA.
19. Eden A. H. & Yahudai A. (1997). Tricks Generate Patterns. *Technical report* 324. Department of Computer Science, Tel Aviv University, Israel.
20. Eden A.H., Gil H. & Yahudai A. (1997). Precise Specification and Automatic Application of Design Patterns. *Proceedings of the 12th IEEE International Automated Software Engineering Conference*, Nevada, USA. pp. 143 – 144.
21 Eden A.H., Yahudai A. & Lundqvist K. (1999). LePUS -Symbolic Logic Modelling of Object Oriented Architectures: a case study. *Proceedings of the Second Nordic Workshop on Software Architecture NOSA99*, Ronneby, Sweden.
22. Mikkonen T. (1998). Formalizing Design Patterns. *Proceedings of the International Conference on Software Engineering*. pp. 115-124.
23. Eden A.H. (1999). Motifs in object oriented architecture. *IEEE Software* **16**(5): 86-93.
24. Eden A. H. (1999). Towards a Mathematical Foundation for Design Patterns. *Technical report 1999-004*. Department of Information Technology, Uppsala University, Sweden.
25. Ambler S. W. (1998). *Process Patterns: Building Large-Scale Systems Using Object Technology*. Cambridge University Press/SIGS Books, UK.
26. Coplien J. O. (1996). *Software Patterns: A White Paper*. SIGS Publications, UK.
27. Coplien J. O. & Zhao L. (2000). Symmetry and Symmetry Breaking in Software Patterns. *Proceeding of the 2nd International Symposium on Generative and Component-Based Software Engineering (GCSE'2000)*, Erfurt, Germany. pp. 37-56.