A Formal Model for Behavioural Properties of Object-Oriented Patterns

Alejandra Cechich(1) and Richard Moore(2)

(1) Department of Informatics and Statistics - University of Comahue
Buenos Aires 1400, 8300 Neuquén, Argentina
E-mail: acechich@uncoma.edu.ar

(2) United Nations University - International Institute for Software Technology
P.O. Box 3058, Macau
E-mail: rm@iist.unu.edu

Abstract

Object-Oriented patterns represent abstractions of good solutions to recurring problems in object-oriented software design. Patterns are described informally in the literature, generally using natural language together with some sort of graphical notation, which makes it difficult to give any meaningful certification of software developed using them. A formal basis for object-oriented patterns which can form the basis for demonstrating that a particular design conforms to a given pattern, has been presented. However, some behavioural properties need to be added to include information relating to the implementation of individual operations in terms of another operations. In this paper, we describe an extension of a previous formal model of patterns in order to include additional behavioural properties, and we illustrate how an instantiation can be done. We also briefly discuss future work, which will consider the formalisation of variations of patterns.

Key words: Object-Oriented design patterns – Formal Methods

Introduction

In object-oriented design methods, design patterns are becoming increasingly popular as a way of identifying and abstracting the key aspects of commonly occurring design structures, and thus as a basis for reusable object-oriented design. Each design pattern lets some aspects of the system structure vary independently of other aspects, which makes the system more robust to changes. In addition, the degree of abstraction offered by the patterns helps designers to identify similar structures in different applications, which leads to more rapid understanding of different applications and hence to applications being built faster.

The GoF patterns [1] provide an infrastructure that defines the components to be included in each solution and how they should be interpreted. However, the GoF patterns and their properties are specified using a combination of graphical notation and natural language, together with sample code in some object-oriented programming language. The description of the patterns is thus largely informal, which makes it difficult to be certain that the patterns themselves are meaningful and contain no inconsistencies and, more importantly, that the pattern is being used correctly and consistently by developers. It is therefore extremely difficult to give any meaningful certification of the correctness of software developed using patterns.

In order to alleviate these problems, a more formal basis for patterns is needed. A first piece of work in this direction [2] represents patterns as formulae in LePUS, a language defined as a small fragment of higher order monadic logic (HOML) [3]. Another work in this direction [4] specifies patterns using DisCo, a language based on Temporal Logic of Actions. Our first work in this direction has presented a preliminary formal model of the essential elements of GoF patterns which can serve as the basis for checking the internal consistency of pattern structures [5]. We have formally specified the general constraints that the static structure and properties of a pattern must satisfy if it is to be meaningful and consistent. But also our model focus on the abstraction of reusable components of patterns as a main feature – in that way other kind of patterns could be specified using the same model.

This work is a first step towards formally verifying the consistency and correctness of the development of an object-oriented design using patterns. However, object-oriented designs can also include information, which cannot be expressed in this model. This includes, for example, information relating to the implementation of individual operations in terms of other operations, which may be stated in the comments in a design or which may even be implicit in the names of the operations; and information about the semantics of the individual classes in the design, and more especially about their relationships and the functionality of their operations (for example that the aggregation relation between the abstraction and the implementor classes in the `Bridge' pattern represents an implementation whereas that between the context and state classes in the `State' pattern represents a link to a common interface for many different behaviours).

In this paper we present an extension of our previous formal model of patterns [5] to include additional behavioural properties. In Section 2 of the paper we introduce the basic components of GoF patterns and we present an overview of our formal model (written in RSL [6]). Section 3 presents our extension of the model to include additional behavioural properties of patterns and also formalises constraints on their various components applying them to some examples on patterns. Future work will modify and extend the model to include variations of a given pattern. This is discussed briefly in the final section of this paper.

2. A Formal Basis for Patterns

2.1 GoF Patterns

In general, an object-oriented pattern has four essential elements:

- The *pattern name*, which is simply a name used to identify the particular type of design problem the pattern addresses;
- The *problem*, which explains the problem the pattern is intended to address and when to apply the pattern. It might also include a set of conditions that must be satisfied before the pattern is applied.
- The *solution*, which describes the components that make up the pattern together with their responsibilities and the relationships and collaborations between them. The solution is abstract, in the sense that it represents a design template which can be applied in a range of different situations.
- The *consequences*, which describes the results and trade-offs that can be expected from applying the pattern.

GoF patterns are presented [1] in terms of thirteen components, which are mostly specialisations of parts of the above elements, such as intent, motivation, applicability, structure, collaborations, participants, consequences, etc. For the preliminary model, we focus primarily on the *solution* component of the pattern, that is its structure, participants and collaborations, since this is the most suitable part to be formalised and is also sufficient to allow us to define the static constraints which are necessary to ensure that a pattern is consistent and meaningful. We deal with the various elements of the solution in more detail below.

Structure. The structure of a GoF pattern is represented as an OMT [7] diagram, which shows the classes which make up the pattern and the relationships between them, and which can also contain comments. As an example, the structure of the `Bridge' pattern, whose intent is to decouple an abstraction from its implementation so that the two can vary independently, is shown in Figure1.

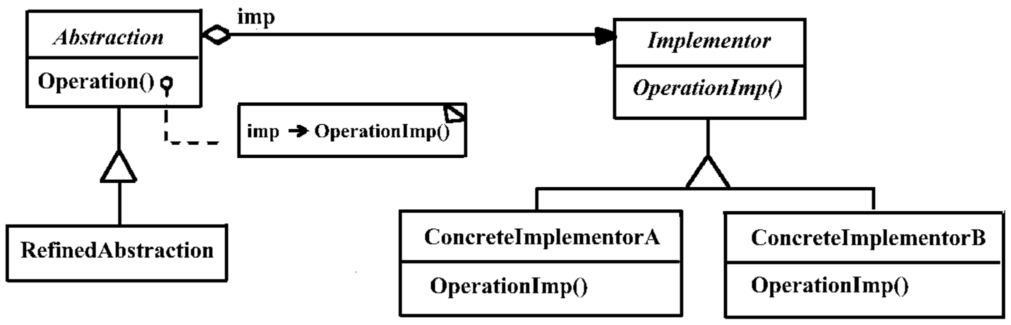

Figure 1 – Bridge Structure

A class specifies an object's internal data and representation and defines (the signatures of) the operations that the object can perform. An operation signature comprises the operation's name and any appropriate parameters required by the operation. The set of all signatures defined by an object is called the *interface* to the object. Any request that matches a signature in the object's interface may be sent to the object. A class name or operation name, which is written in italic script, means that that class or operation is *abstract*, while upright script means it is *concrete*.

A relationship specifies a connection or communication between classes. In the OMT diagram, relationships are represented as lines linking classes. Four different types of relationship are distinguished – inheritance, aggregation, instantiation, and association. Comments also appear as rectangles in the OMT diagram defining the pattern's structure. These rectangles are attached to the class to which they refer by dashed lines which end in a small circle within the class description rectangle. The text of the comment appears in normal face type. The `Bridge' pattern contains one comment which indicates that the operation called `Operation' in the `Abstraction' class is implemented by invoking the operation `OperationImp' in the object `imp', which is the object of the `Implementor' class which is referenced by the `Abstraction' class.

Participants. The participants component of a GoF pattern generally takes the form of a description of the roles and responsibilities of each of the classes in the pattern. For example, the participants of the `Bridge' pattern are:

> **Abstraction**: defines the abstraction's interface and maintains a reference to an object of type Implementor;
> **RefinedAbstraction**: extends the interface defined by the Abstraction class;
> **Implementor**: defines the interface for implementation classes, which can be different from the interface to Abstraction;
> **ConcreteImplementor**: implements the Implementor interface and defines its concrete Implementation.

Collaborations. Collaborations in a GoF pattern are generally represented using an interaction diagram together with some explanatory text, though when the collaborations are very simple, as is the case in the `Bridge' pattern, the textual description is generally sufficient and the diagram is omitted.

Collaborations describe how the pattern's participants interact to carry out their responsibilities. Each collaboration corresponds to an invocation or an instantiation relationship between two pattern classes: the *sender* and the *receiver*. The sender invokes a signature in order to accomplish its responsibilities and the receiver responds by returning a message or by invoking other objects. Alternatively, the sender sends an instantiation message and the receiver creates an object in response. For the `Bridge' pattern, only one collaboration is defined*: `Abstraction' forwards client requests to its `Implementor' object*.

2.2 GoF Patterns in RSL

In the following specifications, knowledge about RSL language or another similar formal language is assumed.

Our formal description of GoF patterns [5] is essentially based on abstractions of three main elements: the *pattern head*, which identifies the particular pattern; the pattern structure; and the pattern collaborations. We have defined a GoF pattern in RSL as a product type composed of these three elements. The pattern head embodies the pattern name and classification (for example, for the `Bridge' pattern the name is Bridge and the classification is object-structural). The other two elements – pattern structure and pattern collaborations – represent the *solution* described by the pattern. We model a pattern structure formally as a set of (well-formed) classes and a set of (well-formed) relationships between them.

GoF_Pattern = G. Pattern_Head X PS.WF_Pattern_Structure X CO.WF_Colls

Pattern_Structure = C.Wf_Class-**set** X R.Wf_Relation-**set**

A class has a name and a set of variables and constants that describe its state. It also contains a set of signatures representing its operations. Finally, it has a type, which may be concrete or abstract as explained in Section 2.1, and a *role*, which abstractly represents its responsibilities. Each operation signature consists of a *signature head*, which specifies the operation's name and the objects it takes as parameters, and the operation's type, which is defined or implemented according to whether the operation is abstract (i.e. unimplemented) or concrete (i.e. implemented) respectively.

A relationship between classes consists of the type of the relation (inheritance, aggregation, association, or instantiation), the relation's name, and the pair of classes it relates. In addition, each of these classes has an associated cardinality whose values, *many* or *one*, are sufficient to model GoF patterns.

```
Pattern_Class :: class_name : G.Name
                 class_state : G.State
                 class_interface : P.Signature-set
                 class_type : G.Class_Type
                 p_role : G.Role

Pattern_Relation ::  related_classes: Related_Classes
                     signature_relation: G.Name
                     relation_type: G.Relation_Type
```

We impose some well-formedness constraints on pattern relations and pattern classes by using RSL subtype definitions. Full details can be found in [8].

There are also constraints on the structure based on the particular responsibilities and roles of the various classes in the pattern. In analysing these, however, we notice that different patterns often have responsibilities which are very similar at an abstract level. For example, the `Bridge', `State' and `Strategy' patterns all have essentially the same basic form: two abstract classes connected by an aggregation relation, with an object of one class having the responsibility of maintaining a reference to an object of the other and with subclasses implementing defined interfaces.

In our model, we abstract these common responsibilities, which we call *common features*, in order to obtain a more general and reusable specification. We illustrate the abstraction by specifying some of the properties of the participants in the `Bridge' and the `State' patterns. In the `Bridge' pattern, classes play 5 different types of role: abstraction, implementor, refined abstraction, concrete implementor, and client. However, in any structure which corresponds to the `Bridge' pattern, there must be exactly one class which plays the abstraction role and exactly one class which plays the implementor role. In a concrete specification of these properties, we might define a variant type, Role_Type, to represent the 5 different types of roles played by classes in the `Bridge' pattern, together with a function which determines the role type from a class' role and two other functions stating respectively that there is only one abstraction class and only one implementor class.

Applying the same approach to the ‘State’ pattern would yield a similar specification, though with different role types and with constraints that only one class plays the context role and one the state role. We can exploit this similarity by using a parameter to abstract these role types and a generic function *exists_one* which tests whether only one class of a given role type exists in a pattern structure:

exists_one: PR.Role_Type $\to$ PS.Pattern_Structure $\to$ **Bool**
exists_one(r) (psc, psr) $\equiv$ ($\exists!$ c: C.Wf_Class • c $\in$ psc $\wedge$ (role_type(C.p_role(c)) = r)

Then, for example, the function *exists_one_abstraction* required in the specification of the 'Bridge' pattern can be written simply in terms of the function *exists_one* applied to the abstraction role:

```
exists_one_abstraction : PS.WF_Pattern_Structure   →  **Bool**
exists_one_abstraction(ps)  ≡  exists_one( B.abstraction )(ps)
```

where B is an object which defines the specific roles for the 'Bridge' pattern.

Further common features are identified and defined in the same way in [8], which also illustrates their use in 5 different patterns.

Following our informal discussion of collaborations in Section 2.1, a collaboration essentially consists of a pair of objects which are communicating, that is the sender and the receiver in the communication, and the message that passes between them, which is effectively the name of the operation that should be invoked in the receiver together with any appropriate parameters. The message thus corresponds to our *Signature_Head* type, and we therefore model a single collaboration as a record type comprising two objects and a signature head:

```
Collaboration::
        sender_o : G.Concrete_Object
        receiver_o : G.Concrete_Object
        signature_coll : P.Signature_Head,
```

Each collaboration thus has a (possibly empty) set of *prerequisites*, which are collaborations which must be executed before it. We therefore introduce the notion of a c*ollaboration identifier* to distinguish individual collaborations in a pattern and model the prerequisites of a collaboration as a set of these collaboration identifiers. Then the record type *Coll* is used to specify a collaboration and its prerequisites, and a mapping between collaboration identifiers and these is used to model all the collaborations in a pattern.

```
Coll :: col : Collaboration   prereq : G.Coll_Id-set,

Collaborations = G.Coll_Id  →  Coll
                            m
```

There are also constraints arising from the consistency between the collaborations and the roles of the classes in the pattern, but just as there are similarities at the abstract level between the responsibilities of different patterns, so there are corresponding similarities amongst their collaborations. For example, the collaborations in the `Bridge', `State' and `Strategy' patterns comprise a basic sequence of two interactions: first, an invocation of an abstraction (respectively context) by a client, and second an invocation by the abstraction (respectively context) of an implementor (respectively state or strategy). In each case, the receiver of the first invocation holds a reference to the receiver of the second invocation, and particular signatures, which are different for the three patterns, are invoked in the collaborations.

We abstract these similarities in our formal model by defining the general function which is parameterised with the three roles of the classes involved in the collaborations (for the `Bridge' pattern these are client, abstraction and implementor), the two signatures invoked (operation and operationImp for the `Bridge' pattern), and the reference required for the second collaboration (imp for the `Bridge' pattern). Full details can be found in [8].

3. A Formal Model for Behavioural Properties

The formal model introduced in the previous section explicitly represents collaborations associated to a pattern. The specification is based on the abstraction of the objects that take part in every collaboration, and the message passed between them. However, this formal description of collaborations does not include additional information relating to behavioural properties of patterns. For example, information relating to the implementation of individual operations in terms of other operations, which may be stated in the comments in a design (see for example the comment attached to the `Window' class in the example discussed in [1] for the 'Bridge' pattern which effectively states that the operation `DrawRect' is implemented by invoking the operation DevDrawLine' four times on the object referenced by the state variable of type `imp') or which may even be implicit in the names of the operations (for instance that `DrawText' is implemented by simply invoking `DevDrawText' in the same example), and information about the semantics of the individual classes in the design, and more especially about their relationships and the functionality of their operations (for example that the aggregation relation between the abstraction and the implementor classes in the `Bridge' pattern represents an implementation whereas that between the context and state classes in the `State' pattern represents a link to a common interface for many different behaviours).

A collaboration can be abstracted as part of the structure of a pattern, and implicitly it can represent more information than our current model. Therefore, a GoF pattern can be modelled as a product type composed of two elements. The *pattern head* identifies a particular pattern; and the *pattern structure* embodies the information related to the structure, participants, and collaborations in the pattern.

```
GoF_Pattern = G. Pattern_Head   X   PS.WF_Pattern_Structure
```

The structure is again formally modelled as a set of classes and a set of relationships. A pattern class has a name, a set of variables that describes its state, a type, which may be concrete or abstract as explained in Section 2.1, and a role. But the operations, previously declared as a set of signatures, are declared now as a set of methods.

```
Pattern_Class :: class_name : G.Name
                 class_state : G.State
                 class_methods : P.Method-set
                 class_type : G.Class_Type
                 p_role : G.Role
```

The methods of a class represent the set of all methods defined by the object's operations. Every method is specified by the operation's signature, the method's body, and the object it returns as a result. The signature consists of a signature head, which specifies the operation's name and the objects it takes as parameters, and the operation's type (defined or implemented). The method body specifies the list of invocations an object should invoke in order to execute, and the change of variables produced as a result of these invocations.

```
Method ::       meth_sig: Signature
                meth_bod: Method_Body
                meth_res: G.Vble

Method_Body:: meth_invok: Meth_Call*
              meth_vbles: G.Vble  $\xrightarrow[m]{}$ G.Vble X Signature_Head
```

A variable change is modelled by a mapping between the updated variable and a product type composed of two elements: the invoked object and the signature that has produced the change. This object is abstractly represented as a variable to facilitate the verification of consistency properties in the model. An abstract definition, *G.Vble*, has been used to represent objects as well as state variables in the model which is enough to abstract both components at this level by using a function for distinguishing them. Further extensions will need a more detailed specification.

Finally, an invocation, *Meth_Call,* is modelled as a reference to the list of invoked objects, *call_vbles*, and the invoked signature in those objects – represented as a *Signature_Head*, which embodies the name of the signature and the objects it takes as parameters.

```
Meth_Call::     call_vbles: G.Vble*
                call_sig: Signature_Head
```

The condition that all operations in a concrete class must be concrete, is introduced as part of a predicated (*is_wf_class*) which is then used to define an RSL subtype representing well-formed classes, namely those which satisfy this predicate. The predicate also includes the constraint that no two operations in a class have the same name, and additional constraints associated with the new model. For example, the set of variables that reference the set of invoked objects must be included in the state of the class (*call_vbles_in_state*); the updated variables must be included in the state of the class (*changed_vble_in_state*); the object that receives the invocation from the method body must be included in the set of invoked objects, and therefore in the state of the class (*receiver_in_call_vbles*); and a defined signature is specified only by its names and parameters (*empty_defined_signature*).

```
call_vbles_in_state: Pattern_Class → Bool
call_vbles_in_state( c ) ≡
        (∀ m: P.Method • m ∈ class_methods( c ) ⇒
                let l = P.meth_invok(P.meth_bod(m)) in
                        ∀ i: Nat • i ∈ inds l ⇒ elems P.call_vbles(l(i)) ⊆ class_state( c )
                end)
```

```
changed_vble_in_state: Pattern_Class  → Bool
changed_vble_in_state( c ) ≡     ( ∀ m: P.Method  •  m  ∈  class_methods( c ) ⇒
                                        dom P.meth_vbles(P.meth_bod(m)) ∈  class_state( c ) )


receiver_in_call_vbles: Pattern_Class → Bool
receiver_in_call_vbles( c ) ≡
    ( ∀ m: P.Method, va: G.Vble  •  m  ∈  class_methods( c )  ∧
        va ∈  dom P.meth_vbles(P.meth_bod(m)) ⇒
          let (v,sh) = P.meth_vbles(P.meth_bod(m))(va) in
              let l = P.meth_invok(P.meth_bod(m)) in
                  ∃ i: Nat • i ∈  inds l ∧ v ∈  elems P.call_vbles(l(i)) ∧ sh = P.call_sig(l(i))
              end
          end)

empty_defined_sig: Pattern_Class → Bool
empty_defined_sig( c ) ≡
        (∀ m: P.Method •  m  ∈  class_methods( c )  ∧
        P.interface_type(P.meth_sig(m)) = G.defined ⇒  empty_method(m))
```

The subtype Wf_Class embodies the above properties:

```
Wf_Class = {| a: Pattern_Class • is_wf_class(a) |}

is_wf_class: Pattern_Class → Bool
is_wf_class(a) ≡ all_implemented(a) ∧  no_duplicated_signatures(a) ∧ call_vbles_in_state(a) ∧
              changed_vble_in_state(a) ∧ receiver_in_call_vbles(a) ∧ empty_defined_sig(a)
```

In similar way, the constraints that the classes and relations in the structure must satisfy in order for the whole structure to be well-defined, are defined; as well as the constraints based on particular responsibilities and roles in the pattern.

The collaborations are now defined in terms of implicit invocations specified inside the classes. In that way, we can specify properties inside the body of the methods associated to a signature of a particular type, and abstract the behaviour behind the patterns. For example, the collaborations of the 'Bridge' pattern are parameterised with the two roles of the classes involved (abstraction, and concreteImplementor), the two signatures invoked (operation and operationImp), and the reference required for the second collaboration (imp). The function *foward_coll* therefore represents all collaborations in which a signature whose type is s1 in an object of role type r1 invokes a signature whose type is s2 in an object of role type r2, which is referenced in its state by a variable of type v. The invocation from s1 to s2 is defined in the body of the method associated to s1 (m1).

We use this function to describe the properties of the collaborations in a particular pattern by giving appropriate values for the role, signature, and variable types in its parameters. For the 'Bridge' pattern, a signature of type 'Operation' in the class whose role is 'Abstraction' has an associated method that invokes at least one signature of type 'OperationImp' in an object of type 'imp' referenced in the state of the class of type ‘Abstraction’. The referenced object corresponds to an instance of the ‘ConcreteImplementor’ class.

In similar way, collaborations in other patterns can be modelled by using the function *forward_coll* , for example in the ‘State’ pattern a ‘Context’ forwards a requirement to its state object, in the ‘Strategy’ pattern a ‘Context’ forwards a requirement to its strategy object, etc.

```
forward_coll:
    PR.Role_Type X PR.Role_Type X PR.Sig_Type X PR.Sig_Type X PR.Vble_Type →
    PS.Pattern_Structure → Bool

foward_coll(r1, r2, s1, s2, v)(psc, psr) ≡
        ( ∀ c1, c2: C.Wf_Class • c1 ∈ psc ∧ c2 ∈ psc ∧
                role_type(C.p_role(c1)) = r1 ∧ role_type(C.p_role(c2)) = r2 ⇒
                  ( ∀ m1, m2: P.Method, va: G.Vble, sh: P.Signature_Head •
                        method_in_class(m1, c1) ∧ method_in_class(m2, c2) ∧
                        sig_in_method(s1, m1) ∧ sig_in_method(s2, m2) ∧
                        P.meth_invok(P.meth_bod(m1)) = ⟨P.mk_Meth_call(⟨va⟩,sh)⟩ ∧
                        s2 = signature_type(P.sig_name(sh)) ∧ vble_type(va) = v))
```

Some auxiliary functions are used in this definition: *method_in_class* checks that the method is inside a given class, and *sig_in_method* checks that a signature inside a given method has a particular type.

For the ‘Bridge’ pattern, the appropriate values for the role type, signatures and variable types are:

```
bridge_coll: PS.WF_Pattern_Structure → Bool
bridge_coll(ps) ≡
  forward_coll(B.abstraction, B.concreteIimplementor, B.operation, B.operationImp, B.imp)(ps)
```

Other functions defining properties of collaborations at a level of abstraction similar to that of *forward_coll,* including one which checks the result of executing a method and another which models repeated invocations like those in the ‘Observer’ pattern, are defined in [9].

Conclusions

In this paper, we have presented an abstract formal model of the essential elements of GoF patterns written in RSL, and we have formalised various constraints related to the structure, participants, and collaborations of these patterns. We have modified and extended our previous formal model to allow us to specify additional behavioural properties, and we have illustrated using examples how this can be done.

This is another step towards formally verifying the consistency and correctness of the development of an object-oriented design using patterns. However, object-oriented design can also include variations of the standard patters as presented in [1], which cannot be expressed in our current model. For example, several variations of patterns can be identified in the design of Java libraries [10]: in normal ‘Bridge’, an abstraction calls methods on its implementation; in Java libraries an abstraction often calls methods on its implementation which returns new customised values by using a hierarchical structure based on other patterns. As another example on Java libraries design, a combination of an adapted ‘Mediator’, ‘Observer’, and ‘Adapter’ is used to model applets which supports an interface for each kind of handled event.

In the next stage of our work, we are modifying our formal model to allow us to specify variations of patterns. Formalising semantic information relating to the classes and relationships is more problematical, especially at the abstract level, and we are investigating this issue by carrying out a case study involving the use of design patterns in a real application in the area of health care (information systems for intensive care units).

References

[1] Gamma E. and Helm R. and Johnson R. and Vlissides J.:
Design Patterns – Elements of Reusable Object-Oriented Software,
Addison-Wesley (1995)

[2] Eden A. and Gil J. and Hirshfeld Y. and Yehudai A.:
Towards a Mathematical Foundation for Design Patterns,
http://www.math.tau.ac.il/~eden/bibliography

[3] Eden A. and Hirshfeld Y. and Yehudai A.:
LePus – A Declarative Pattern Specification Language
http://www.math.tau.ac.il/~eden/bibliography

[4] Mikkonen Tommi:
Formalizing Design Patterns
Proceedings of the 20th International Conference on Software Engineering
IEEE Computer Society Press (1998) 115-124

[5] Cechich A. and Moore R.:
A Formal Basis for Object-Oriented Patterns
Proceedings of the 6th Asia-Pacific Software Engineering Conference
IEEE Computer Society Press (1999) 284-291

[6] The RAISE Language Group:
The RAISE SPECIFICATION LANGUAGE,
Prentice-Hall (1992)

[7] Rumbaugh J.:
Object-Oriented Modeling and Design:
Prentice-Hall (1990)

[8] Cechich A. and Moore R.:
A Formal Specification of GoF Design Patterns
UNU/IIST Technical Report 151, January 1999
http://www.iist.unu.edu

[9] Cechich A. and Moore R.:
An Extended Model of Collaborations
UNC/AIS Technical Report 4, January 2000

[10] Java Patterns:
http://st-www.cs.uiuc.edu/cgi-bin/wikic/JavaAWT