



The United Nations  
University

**UNU-IIST**

International Institute for  
Software Technology

---

# Towards a Calculus for Design Pat- terns

---

Abdel Hakim Hannousse and Zhiming Liu

October 2007

## UNU-IIST and UNU-IIST Reports

UNU-IIST (United Nations University International Institute for Software Technology) is a Research and Training Centre of the United Nations University (UNU). It is based in Macao, and was founded in 1991. It started operations in July 1992. UNU-IIST is jointly funded by the government of Macao and the governments of the People's Republic of China and Portugal through a contribution to the UNU Endowment Fund. As well as providing two-thirds of the endowment fund, the Macao authorities also supply UNU-IIST with its office premises and furniture and subsidise fellow accommodation.

The mission of UNU-IIST is to assist developing countries in the application and development of software technology.

UNU-IIST contributes through its programmatic activities:

1. Advanced development projects, in which software techniques supported by tools are applied,
2. Research projects, in which new techniques for software development are investigated,
3. Curriculum development projects, in which courses of software technology for universities in developing countries are developed,
4. University development projects, which complement the curriculum development projects by aiming to strengthen all aspects of computer science teaching in universities in developing countries,
5. Schools and Courses, which typically teach advanced software development techniques,
6. Events, in which conferences and workshops are organised or supported by UNU-IIST, and
7. Dissemination, in which UNU-IIST regularly distributes to developing countries information on international progress of software technology.

Fellows, who are young scientists and engineers from developing countries, are invited to actively participate in all these projects. By doing the projects they are trained.

At present, the technical focus of UNU-IIST is on **formal methods** for software development. UNU-IIST is an internationally recognised center in the area of formal methods. However, no software technique is universally applicable. We are prepared to choose complementary techniques for our projects, if necessary.

UNU-IIST produces a report series. Reports are either Research R, Technical T, Compendia C or Administrative A. They are records of UNU-IIST activities and research and development achievements. Many of the reports are also published in conference proceedings and journals.

Please write to UNU-IIST at P.O. Box 3058, Macao or visit UNU-IIST's home page: <http://www.iist.unu.edu>, if you would like to know more about UNU-IIST and its report series.

G. M. Reed, Director



The United Nations  
University

**UNU-IIST**

**International Institute for  
Software Technology**

P.O. Box 3058  
Macao

---

# Towards a Calculus for Design Patterns

---

**Abdel Hakim Hannousse and Zhiming Liu**

## **Abstract**

The overarching goal of a formal specification of design patterns is to enhance their understandability, address their composability problem and provide a tool support for their re-usability. Understanding design patterns means uncovering when and how a specific pattern can be used to resolve a specific design problem. The lack of completeness of the existing formal specifications motivates this work. In this paper we start introducing a calculus for design patterns based on rCOS (Refinement Calculus of Object and Component Systems) specification which provides a formalization of patterns addresses the pattern composition mechanism and describes a mechanism for design refinement to patterns designs. In this first step we are focusing on the proposition of a formalization model. Thus, a model for pattern's formalization is proposed and an assessment model is used for the evaluation of the proposed model and comparing it with the existing ones to show its efficiencies and limitations.

**Abdel Hakim Hannousse** is a fellow of UNU-IIST from Mohammed Khider University, Biskra, Algeria, where he is a doctoral candidate. His research interests include formal specification and verification, aspect oriented software development, multi-agent system modeling, robotics and design patterns. His email address is `hakim@iist.unu.edu`, `hannousse_a_hakim@yahoo.fr`.

**Zhiming Liu** is a Research Fellow at UNU-IIST. His research interests include theory of computing systems, emphasizing sound methods for specification, verification and refinement of fault-tolerant, realtime and concurrent systems, and formal techniques for OO development. His teaching interests are Communication, Concurrent and Distributed Programming, Internet Security, Software Engineering, Formal specification and Design of Computer Systems. His email address is `Z.Liu@iist.unu.edu`

---

## Contents

<b>1</b>	<b>Motivation</b>	<b>1</b>
<b>2</b>	<b>Related Work</b>	<b>2</b>
<b>3</b>	<b>Design Patterns on rCOS Specification</b>	<b>4</b>
3.1	Object Oriented Design Model . . . . .	5
3.2	Structure Variables . . . . .	7
<b>4</b>	<b>Patterns and Pattern Specifications</b>	<b>8</b>
4.1	Adapter Pattern . . . . .	8
4.2	Strategy Pattern . . . . .	9
4.3	Observer Pattern . . . . .	10
4.4	Decorator Pattern . . . . .	11
4.5	Composite Pattern . . . . .	13
4.6	Abstract Factory Pattern . . . . .	14
4.7	Singleton Pattern . . . . .	16
<b>5</b>	<b>Assessment and Comparison</b>	<b>16</b>
5.1	The Assessment criteria . . . . .	16
5.1.1	Formalism . . . . .	17
5.1.2	Completeness . . . . .	17
5.1.3	Ease-of-Use . . . . .	17
5.2	The Assessment results . . . . .	17
<b>6</b>	<b>Conclusion &amp; Future Work</b>	<b>19</b>



# 1 Motivation

The usability of design patterns is inevitably growing since the arising of the GoF catalog [7]. In fact, design patterns effects can be shown through the minimization of the design efforts and the facilitation of understanding existing designs. However, an informal specification of patterns has some limitations. Lack of precision, object oriented related notation, and cataloging issues are some of those limitations. Since the theoretical proof of the feasibility of formalizing design patterns [16], many researches are dedicated whether to use existing specification languages such as B[11], RSL[4] for patterns specification or to develop appropriate languages such as ELePUS[2] and BPSL[1]. Unfortunately, effective specification of patterns cannot be shown through the above alternatives. That is due to the fact of the limitations they have whether in the logic used for the specification or in the specification of illegal parts of the existing textual descriptions. Our work aims to propose a calculus for design patterns which satisfy the following points:

- Formalize design patterns in order to clarify their aim and usability.
- Formalize pattern composition for correctly developing application designs directly using pattern designs as building blocks.
- Enhance the quality of existing designs by studying design refinement to pattern designs.
- Finally, develop a tool support for automatic or semi-automatic satisfaction of the above points.

In our mind, a complete and useful formal specification of patterns cannot be reached by specifying only their structures and/or behaviors. A complete and a useful one should clarify *when* and *how* a pattern can be used. The specification of the structure and the behavior of patterns describes only the *how* aspect of patterns which does not tell anything about where a specific pattern can be used. In fact, both *how* and *when* are complementary aspects for any formal pattern specification. While the *when* aspect describes the problem and the context under which a pattern can be used, the *how* aspect describes the solution provided by a pattern to resolve that problem. Our idea has been inspired from the common definition of a pattern.

*"Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution."* [31]

Accordingly, an accurate formalization of a pattern should include the specification of the three main parts, the problem, solution and the context and describes the relationships between them. In our work we consider that relation as a refinement relation from a problem to its solution when the context holds.

*problem*  $\sqsubseteq$  *solution*    **if** *context*

Achieving our goal, this paper is divided into three main parts: A presentation of the existing formal specification of patterns with their limitations, a description of our proposal for formalizing patterns. Finally an assessment comparison is made to show the advantages and the disadvantages of our proposal.

## 2 Related Work

Bayley and Zhu[5] proposed a method for formalizing patterns using a first-order logical predicates. Their proposed approach consists of achieving two main steps. Firstly, identifying classes, operations and associations used in the structure part of patterns. Secondly, state conditions that must be applied to them both in English and in predicate logic. However, describing just the structure part of a pattern through its class diagram is not enough to describe the semantic of that pattern. The behavioral part (i.e. the sequence diagram) explains how the structural part react to achieve the attended behavior of such pattern.

Taibi and Ling[1] proposed a balanced approach for patterns specification. That approach consists of specifying both the structural and the behavioral aspects of patterns using the first-order logic and the temporal logic of actions respectively. The basic idea presented in that work consists of decomposing a program to primary entities (i.e. classes, methods, attributes, objects and untyped values) and defining primary permanent relations as predicates acting upon those primary entities. The temporal logic is used to describe the behavioral aspect of patterns by defining actions that change variable states, also to associate and disassociate objects through temporal relations. Unfortunately, this work is focused only on the specification of the solution part of the pattern and not on other parts such as the intent and the applicability parts. This is a common problem for the most current alternative specifications, where only the *how* aspect is considered and the *when* aspect is omitted.

Sivakumar[2], proposed an extension of the LePUS language[6] for making it able to specify the intent and the applicability parts of patterns. ELePUS uses the first-order logic to specify the relationships among a lot of ground variables(i.e. classes, functions), hierarchy variables(i.e. inheritance class hierarchies) and higher dimension variables(i.e. set of entities). The relationships among entities are classified to ground relations(i.e. defined between ground entities), generalized relations (i.e. ground relation applied to many variables) and commuting relations (i.e. specify either the list of regular relations commute). Those relations are presented as predicates in patterns specification. Unfortunately, the specification problem has not been completely resolved. The specification presented is at the higher level abstraction, where the creational patterns need a lower level detail for their specification(i.e. number of objects created, etc.). Also, in that work the different specification parts are presented separately. In our mind the refinement must be used to make relationships between the different specification parts to complete patterns understanding.

Cechich and Moore[9] defined a formal model for GoF patterns[7] based on RSL(RAISE Specification Language). In their proposed model, a pattern is represented as a product type in RSL composed on three elements: a head, a structure and a list of collaborations. The head element describes the name, the purpose and the scope of the pattern. The structure element specifies a set of classes related by a set of RSL relations. The collaboration element describes a list of constraints on the order of operation calls. That work was focused only on the validation problem(i.e. how a given design matches a particular pattern). Also, the specification of the head does not add anything to the existing textual description.

Marcano et al.[11] described a semi-formal method to specify patterns by using the UML and the *B* methods as complementary paradigms. In that sense, the UML refinement is defined and the *B* method is used to describe the semantic of the refined design. The refinement concept is used to describe a transition from a specific problem to its solution by applying patterns specialization. In our mind, using the *B* method for patterns specification preserves the abstraction level of patterns(i.e. object notation limitation). However, the problem identification step presented to apply pattern specialization is not defined formally, they just capture problems in an informal manner, then use the UML refinement and the *B* specification to describe formally the solution.

Blazy and Zhu [10] described a model for reusing the so-called specification patterns, according to the authors, a specification patterns are design patterns that are expressed in a formal language, the formal language used by the author is the B. In that work, a pattern is defined as one single abstract machine. Only the pattern solution is described. The structure of the solution is defined in term of B machine variables and invariants, while the behavior is described as pre and postcondition in the machine operation specification. The instantiation mechanism is also implemented in B by the inclusion of machines, in such way, the machine corresponding to the pattern is included in the machine corresponding to the instantiation of the pattern. Three kind of compositions are introduced. The juxtaposition composition where patterns can be composed without defining any link between them, in this case, the extension mechanism is used to allow all the operations of the underlying patterns to be considered as genuine operations of the composed machine. The composition with inter-patterns links where new relations between the variables of the composed pattern machine can be added. Finally, the unification composition that allows some variables of the composed patterns to be merged. That work has two main limitations. Firstly, a pattern is specified as one single machine which is difficult to read and maintain. Secondly, is a technical problem which consists of defining the new operations of an extension before applying the extension mechanism. Also, combining several instances of the same pattern has not mentioned and the operation composition has not precisely defined.

Mikkonen [20] has proposed a formal specification of design patterns using the DisCo specification method. The DisCo specification method provides a way to describe patterns using classes, relations and actions. Class definitions are formulas defining the general form of possible objects. Relations definitions describe object associations with each others. Actions are temporal logic of action formulas that describes the temporal behavior of objects. Accordingly, a pattern can be formalized in DisCo as a set of class definitions, a set of relation descriptions and temporal actions specifications. Also, a pattern composition mechanism was discussed in that work. According to the author, the nature of DisCo refinement can implement the pattern

composition where the characteristic proprieties of the composed patterns are preserved. The basic idea behind the proposed DisCo formalization and composition consists of providing a rigorous reasoning about patterns communication by formalizing their temporal behaviors at a higher level abstraction, the DisCo refinement mechanism is used to specify pattern composition. Unfortunately, that specification does not describe when a pattern can be applied, it just describes how the structural and the behavioral aspects of patterns can be formalized. Furthermore, the correct composition of patterns in that work depends on the refinement correctness of DisCo in the sense that DisCo refinement preserves the proprieties of the combined patterns. The question here is: can that refinement ensures that no undesirable propriety can appear after the composition process ?

Alencar et al. [21] have presented an interesting idea about how we can use Prolog facts and rules to formalize patterns. In the proposed approach by the authors, a pattern can be described as primitive operator which can be described in terms of primitive object oriented design primitives. The object oriented design primitives are described in a predicate-like format. The instantiation mechanism and structure addition and removal facts provided by Prolog facilitate the integration and the evolution of patterns. However, that work has two limitations. Firstly, the requirements of applying patterns cannot be specified. Secondly, the integration mechanism can provide an inconsistency of the resulting composition which may imply an incorrect composition. Checking the inconsistency by specifying some rules in Prolog is not an efficient way, the user is unable to specify manually all the inconsistency rules of the composed design.

### 3 Design Patterns on rCOS Specification

In order to properly use a pattern in your design, you must uphold three criteria: understand your problem, understand the pattern and understand how the pattern solves your problem. Furthermore, introducing a formalism for design patterns describes them accurately and allows rigorous reasoning about which pattern solves which problem and how the pattern do it. The previous section shown the lack of completeness of the existing formal specifications. In this section we try to define, specify and propose a calculus for patterns and pattern composition using rCOS specification.

In fact, a design pattern can be shown as a structural and/or behavioral *transformation* of an initial, non complete and non re-usable object oriented design model, that the designer needs to satisfy his or her system requirements, to another final, complete and more re-usable one which is in fact a *refinement* of the former under a condition. In the following we describe a *model* for object oriented design based on rCOS and describe the GoF patterns as refinement rules following that model.

### 3.1 Object Oriented Design Model

Our proposed model for object oriented design consists of a structure declaration section *Cdecls*. The main difference between our specification model and that defined in rCOS is that the *Cdecls* used here extends the original one used in rCOS by describing a finite sequence of *abstract classes* and *interfaces*. Furthermore, the *Main* part of the rCOS specification is omitted for the reason that our aim is to specify design patterns and a design pattern is not a complete program as rCOS used for, it is just a prototype in which just a part of its structure and behavior are specified. The behavior of the object oriented design model is described by methods specifications which are specified as ordinary rCOS commands that their semantics is described as a framed design in terms of UTP [25].

Similar to that defined in rCOS, a class declaration has the same structure of that used in the original rCOS specification with the assertion that a class structure can implement a set of interface structures already declared in *Cdecls*. An interface structure is specified as a family of operation signatures as defined in the extended version of rCOS for component systems [24] with the assertion that an interface structure may inherit only an interface structure and cannot implement any other declared structure in *Cdecls*.

An abstract class declaration is the more general form which can include both methods and operation signatures declarations. We call operation signatures abstract methods. In the following we describe the complete rCOS syntax provided to describe our object oriented design model in a like-BNF<sup>1</sup> notation.

$$\begin{aligned}
cdecls \in Cdecls & ::= \emptyset \mid cdecl \ cdecls \\
cdecl \in Cdecl & ::= [\mathbf{abstract} \ \mathbf{class} \ C_1[\mathbf{extends} \ C_2][\mathbf{implements} \ I_1, I_2, \dots, I_3]\{ \\
& \quad \mathbf{private} \quad T_{11} \ a_{11} = d_{11}, \dots, T_{1m_1} \ a_{1m_1} = d_{1m_1}; \\
& \quad \mathbf{public} \quad T_{21} \ a_{21} = d_{21}, \dots, T_{2m_2} \ a_{2m_2} = d_{2m_2}; \\
& \quad \mathbf{method} \quad m_1(T_{11} \ x_1; T_{12} \ y_1; T_{13} \ z_1)\{c_1\}; \\
& \quad \quad \quad \dots; \\
& \quad \quad \quad m_l(T_{l1} \ x_l; T_{l2} \ y_l; T_{l3} \ z_l)\{c_l\}; \\
& \quad \mathbf{signature} \quad m_{l+1}(T_{(l+1)1} \ x_{l+1}; T_{(l+1)2} \ y_{l+1}; T_{(l+1)3} \ z_{l+1}); \\
& \quad \quad \quad \dots; \\
& \quad \quad \quad m_k(T_{k1} \ x_k; T_{k2} \ y_k; T_{k3} \ z_k); ] \\
& \quad \} \\
& \mid \mathbf{interface} \ I_1[\mathbf{extends} \ I_2]\{ \\
& \quad \mathbf{public} \quad T_{11} \ a_{11} = d_{11}, \dots, T_{1m_1} \ a_{1m_1} = d_{1m_1}; \\
& \quad \mathbf{signature} \quad m_1(T_{11} \ x_1; T_{12} \ y_1; T_{13} \ z_1); \\
& \quad \quad \quad \dots; \\
& \quad \quad \quad m_l(T_{l1} \ x_l; T_{l2} \ y_l; T_{l3} \ z_l); \\
& \quad \} \\
c \in cds & ::= \mathbf{skip} \mid \mathbf{chaos} \mid \mathbf{var} \ T \ x[= e] \mid \mathbf{end} \ x \mid c; c \mid c \triangleleft b \triangleright c \mid c \sqcap c \\
& \quad \mid b * c \mid le.m(\underline{e}, \underline{e}) \mid le := e \mid S.\mathbf{new}(le) \\
le \in exp & ::= x \mid a \mid \mathbf{null} \mid \mathbf{self} \mid \mathbf{super} \mid (C)le \mid le.a \mid f(le)
\end{aligned}$$

<sup>1</sup>BNF here stands for the formal notation Backus-Naur Form

According to the above syntax declaration, an object oriented design model is a non ordered sequence of structure declarations. A structure declaration has two alternative declarations, we refer to the first form as class structure declaration and to the second form as an interface structure declaration. In such way, if a structure declaration follows the first form it could be an abstract class structure or a concrete class structure depends on either the optional clause **abstract** is used or not, if it is the case the optional clause **signature** can be appear, that is due to the fact that an abstract class may have both methods and signatures. If the structure declaration follows the second form it will be considered as an interface structure. The optional **extends** clause is used to determine the immediate superstructure of *cdecl*. In its absence, *cdecl* extends an empty structure specified by  $\emptyset$  in the above specification. The optional **implements** clause determines the set of implemented interface structures by *cdecl*. The **pri** and **public** clauses introduce the private and public attributes of *cdecl*. The visibility mechanism is similar to that of Java: private attributes are visible only inside the structure in which it is declared and public attributes are visible to all the other structures in *cdecls*. Follow the **pri** and **public** clauses, there is a set of method declarations that follow the **method** clause. The method introduced by  $m(T_1 x; T_2 y; T_3 z)\{c\}$  is named *m*; its parameters are *x*, *y*, *z* which determine the value parameters, result parameters and value-result parameters, respectively. The method body *c* is described as a sequence of commands. Follow the **method** clause, the optional **signature** clause is used to determine a set of abstract method declarations, this part is used only for abstract class structure declarations, as it is used in Java an abstract method is a method signature without body specification. All the methods in our model are considered to be public. The interface structure declaration is similar to that used for a class structure declaration with the assertion that an interface structure couldn't declare private attributes and it contains only abstract methods. Similar to that used in Java, we use the **interface** to introduce interface structures. In the sequel, we propose to use more simplified notation for a structure declaration *cdecl*. Accordingly, a structure declaration is a model of the form.

$$type[superclass; imf; pub; pri; op; sig]$$

Where *type* is the type of the *cdecl* structure, *type* is a structural variable whose values are  $\{S, C, I\}$  to denote abstract class, concrete class and interface structure, respectively. In order to distinguish the different structures they have the same type in the object oriented design model we associate indexes to types. For example we use  $S_1$  and  $S_2$  to describe two different abstract class structures. *superclass* denotes the name of the super structure that the structure *cdecl* inherit some of its features. *imf* denotes the set of the names of the implemented interface structures by *cdecl*. *pub*, *pri*, *op* and *sig* denote the sets of the public attributes, private attributes, concrete methods and abstract methods declared for the structure declaration *cdecl*, respectively. Where there is no confusion, we only explicitly give the parameters that we are concerned.

### 3.2 Structure Variables

In this section we recall the structure variables defined in rCOS and used for our object oriented design model.

- $cname$ : the set  $\{name(cdecl) \mid cdecl \in cdecls\}$  of a structure declaration section.
- $name(m)$ : denotes the name of a method  $m$ .
- $body(m)$ : denotes the body of a method  $m$ .
- $\zeta(m)$ : denotes the signature of a method  $m$ , the signature of a method  $m$ .
- $attr(cdecl)$ : is the set of the attributes declared in  $cdecl$ .
- $dtype(a)$ : denotes the declared type  $T$  of an attribute  $\langle a : T, d \rangle$ .
- $superclass$ : denote the superclass name of a structure  $cdecl$ .
- $\Pi(cdecl)$ : denotes the current configuration of the program whose value is the set of objects created far from the structure  $cdecl$  where  $cdecl$  is a concrete class structure otherwise  $\Pi(cdecl) = \emptyset$ .

Obviously, not all the object oriented design model structures in a class declaration section  $cdecls$  described above are *well-defined* declarations. Accordingly, the following definition provides some restrictions to ensure the well-definedness of an individual design structure declaration  $cdecl$ , then we generalize it for the well-definedness of the class declaration section  $cdecls$ .

**Definition 1.** A design structure declaration  $cdecl$  is **well-defined** iff the following conditions hold:

1.  $cdecl_2 = superclass(cdecl_1) \Rightarrow \left( \begin{array}{c} cdecl_1 \neq cdecl_2 \wedge \\ \{cdecl_1, cdecl_2\} \subseteq cdecls \wedge \\ (type(cdecl_1) = type(cdecl_2)) \vee \\ (type(cdecl_1) \in \{S, C\} \wedge type(cdecl_2) \in \{S, C\}) \end{array} \right)$
2. the inheritance relation over the set  $cdecls$  is acyclic.
3. any type used in declarations of attributes and parameters is either a primitive built-in type or a class in  $cname$ .
4.  $\forall cdecl' \in imf(cdecl), type(cdecl') = I$ .
5.  $\forall cdecl' \in imf(cdecl), \forall s \in aop(cdecl'), Mspec(s) \in op(cdecl)$ , where  $Mspec$  is a function that associates a specification to each signature and provides a concrete method. It is the same function used in [24] to describe the interface contract for rCOS component systems.
6. the names of the public and private attributes of each structure are distinct.
7. the names of the methods and the signatures of each abstract class are distinct.

In the following we use  $\mathcal{D}(cdecl)$  to denote the conjunction of the conditions for the well-definedness of a design structure  $cdecl$ .

**Definition 2.** A structure declaration  $cdecls = cdecl_1 cdecl_2 \dots cdecl_n$  is **well-defined** iff each individual structure declaration is well-defined and declared only once in  $cdecls$ .

$$\mathcal{D}(cdecls = cdecl_1 cdecl_2 \dots cdecl_n) \stackrel{\text{def}}{=} \left( \begin{array}{l} \forall cdecl, cdecl' \in cdecls, cdecl \neq cdecl' \wedge \\ \mathcal{D}(cdecl_1) \wedge \dots \wedge \mathcal{D}(cdecl_n) \end{array} \right)$$

## 4 Patterns and Pattern Specifications

In the following we try to describe what the design pattern is in term of object oriented design model, then we use that definition for describing a set of the GoF patterns [7]. We start with six patterns: Adapter, Strategy, Observer, Decorator, Composite and Abstract factory patterns. For a given structure declaration  $cdecls$  and a structure declaration predicate  $p$ , we write

$$cdecls \models p,$$

to denote that  $p$  yields *true* when evaluated over  $cdecls$

**Definition 3.** A design pattern  $\rho = (cdecls_1, cdecls_2, p)$  is a tuple consists of two object oriented models  $cdecls_1$  and  $cdecls_2$  describing the design problem and solution of the pattern, respectively; and a structure predicate  $p$  such that:

1.  $cdecls_1 \not\models p$
2.  $cdecls_1 \sqsubseteq cdecls_2$
3.  $cdecls_2 \models p$

### 4.1 Adapter Pattern

The Adapter pattern aim is to convert the interface of a class into another. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces [7]. The problem and the solution parts of the pattern are described in term of object oriented design models  $cdecls_1$  and  $cdecls_2$ , respectively. The context of the pattern is specified by the predicate  $p$  which is not satisfied by  $cdecls_1$  and satisfied by  $cdecls_2$ . The following table describes the specification of the adapter pattern following our model assuming that  $\bar{x}_i$  is the set of attributes handled by the method  $m_i$ . We use the  $\bar{x}$  notation to denote the same set of handled attributes for two different methods  $m_1$  and  $m_2$  implemented by different classes  $C_1$  and  $C_2$ , respectively.

$cdecls_1$	=	$S_1[; sig \cup \{\zeta(m_1)\};]$ $C_1[S_1; pub \cup \bar{x}; op \cup \{m_1\{c\}\};]$ $C_2[; pub \cup \bar{x}; op \cup \{m_2\{c\}\};]$
$p$	=	$\forall C_1, C_2 \in cdecls, \forall m_1 \in op(C_1), \forall m_2 \in op(C_2), body(m_1) = body(m_2) \Rightarrow$ $(name(m_1) = name(m_2)) \vee (\exists a \in attr(C_1), dtype(a) = C_2 \wedge body(m_1) = \{a.m_2\})$
$cdecls_2$	=	$S_1[; sig \cup \{\zeta(m_1)\};]$ $C_1[S_1; pub \cup \bar{x} \cup \{(a, C_2, (a, C_2, \sigma))\}; op \cup \{m_1\{a.m_2\}\};]$ $C_2[; pub \cup \bar{x}; op \cup \{m_2\{c\}\};]$

Table 1: The Adapter Pattern Specification

According to the above specification, the adapter pattern problem is described as an object oriented design model  $cdecls_1$  which consists of an abstract class structure  $S_1$  and two concrete class structures  $C_1$  and  $C_2$ . The class structures  $C_1$  and  $C_2$  implement the methods  $m_1$  and  $m_2$ , respectively, which they have different names with same behavior (i.e. body). The adapter pattern is used to re-use the behavior of classes (i.e. methods) even if they have different interfaces (i.e. method names). This propriety is described in the above specification as a predicate  $p$  which is not satisfied by  $cdecls_1$ . The object oriented design model  $cdecls_2$  describes the solution part provided by the adapter pattern to achieve that aim. the refined object oriented design model  $cdecls_2$  of  $cdecls_1$  introduces a relationship between the class structures  $C_1$  and  $C_2$  by introducing an attribute  $a$  of type  $C_2$  to the set of attributes of  $C_1$ . In addition, the body of  $m_1$  is refined to an object method call  $a.m_2$ , in such way the predicate  $p$  becomes satisfied by  $cdecls_2$ .

The tuple  $(a, C_2, \sigma)$  is used to denote the initial value of the introduced attribute  $a$  follows the definition of the object value in rCOS described in [14], this tuple denotes that the object  $a$  is initially created. Using a mapping function from our specification described by  $cdecls_2$  to the adapter pattern structure and behavior presented in [7] we find that  $S_1$  describes the *Target* interface,  $C_1$  and  $C_2$  describe the *Adapter* and the *Adaptee* classes, respectively. In addition,  $m_1$  and  $m_2$  describe the *request* and the *specificRequest* methods, respectively.

*Proof.* The adapter pattern refinement rule is an instance of replace method with method object refinement rule of rCOS.  $\square$

## 4.2 Strategy Pattern

The strategy pattern is required to provide a way to configure a class with one of many behaviors (i.e. Algorithms). Accordingly, the strategy pattern is applicable when a design model has a class structure which implements a method that has many behaviors and you want to encapsulate each one in a different class. It is claimed explicitly in [7] that a code containing complicated conditional statements often indicates the need to apply the strategy pattern. Accordingly, The following table describes the specification of the strategy pattern assuming that  $\bar{x}_1$  and  $\bar{x}_2$  denotes the set of public attributes handled by  $c_A$  and  $c_B$ , respectively, and  $\bar{x} = \bar{x}_1 \cup \bar{x}_2$ , where  $\bar{x}_1$  and  $\bar{x}_2$  may contain common elements.

$cdecls_1$	$= C_1[; pub \cup \bar{x}; op \cup \{m_1\{c_A \triangleleft b \triangleright c_B\}\};]$
$p$	$= \exists C_1[; op \cup \{m_1\{c_A\}\};], C_2[; op \cup \{m_2\{c_B\}\};] \in cdecls$
$cdecls_2$	$= S_1[; sig \cup \{s(m_2)\};]$ $C_1[; pub \cup \bar{x} \cup \{(a, S_1, null)\};]$ $op \cup \{m_1\{C_{21}.new(a)[\forall x \in \bar{x}_1, a.x = self.x] \triangleleft b \triangleright C_{22}.new(a)[\forall x \in \bar{x}_2, a.x = self.x]; a.m_2\}\};]$ $C_{11}[S_1; pub \cup \bar{x}_1; op \cup \{m_2\{c_A\}\};]$ $C_{12}[S_1; pub \cup \bar{x}_2; op \cup \{m_2\{c_B\}\};]$

Table 2: The Strategy Pattern Specification

The above specification shows how a design that contains a class structure  $C_1$  which has a method  $m_1$  with conditional statement can be simplified by refining it to a simple one that contains an abstract class structure  $S_1$  with two concrete subclass structures  $C_{11}$  and  $C_{12}$  each of which implements one of the behaviors provided by the original method  $m_1$ . The attributes used by  $c_A$  and  $c_B$  have been declared in  $C_{11}$  and  $C_{12}$ , respectively, and initialized to reference the corresponding ones in  $C_1$  at the constructor execution. This feature has been denoted between brackets follow the object creation command. By mapping to the description of the strategy pattern described in [7] we find that  $C_1, S_1, C_{11}$  and  $C_{12}$  correspond to *Context, Strategy, concreteStrategyA* and *concreteStrategyB*, respectively.

*Proof.* Instance from replace conditional choice with polymorphism refinement rule in rCOS.  $\square$

### 4.3 Observer Pattern

The observer pattern is used to define a one-to-many dependency between objects. So that, when one object changes state all its dependents are notified and updated automatically. Accordingly, the following table describes the specification of the observer pattern following our model.

$cdecls_1$	$= C_1[; pub \cup \{(a, T, v)\}; op \cup \{m_1\{c_1; a = le; c_2\}\};]$ $C_2[; pub \cup \{(b, T, v)\}; op;]$ $C_3[; pub \cup \{(c, T, v)\}; op;]$
$p$	$= \forall o_1 \in \Pi(C_1), \forall o_2 \in \Pi(C_2), \forall o_3 \in \Pi(C_3), o_1.a = o_2.b = o_3.c$
$cdecls_2$	$= S_1[; pub \cup s = \{(o_2, S_2, (o_2, C_2, \sigma)), (o_3, S_2, (o_3, C_3, \sigma))\}; op \cup \{m_2\{\forall o \in s, o.m_3\}\};]$ $S_2[; sig \cup \{s(m_3)\}]$ $C_1[S_1; pub \cup \{(a, T, v)\}; op \cup \{m_1\{c_1; a = le; c_2; m_2\}\};]$ $C_2[S_2; pub \cup \{(b, T, v)\} \cup \{(o_1, C_1, \sigma)\}; op \cup \{m_3\{b = o_1.a\}\};]$ $C_3[S_2; pub \cup \{(c, T, v)\} \cup \{(o_1, C_1, \sigma)\}; op \cup \{m_3\{c = o_1.a\}\};]$

Table 3: The Observer Pattern Specification

In the proposed specification, the state of an object  $o_1$  of the class structure  $C_1$  is specified as a simple attribute  $a$ . This specification shows when a method  $m_1 \in op(C_1)$  changes the value of the attribute  $a$ , the other objects of  $C_1$  and  $C_2$  should make a copy of the new  $o_1$  state by changing their corresponding attributes values  $b$  and  $c$ , respectively, to that of  $a$ . The refinement rule describing the observer pattern is not primitive in rCOS<sup>2</sup>, but it can be proved

<sup>2</sup>All the above proved rCOS refinement laws have been considered as primitive laws, the laws denoted with \*

using deductive reasoning through the rCOS primitive laws by applying a sequence of rCOS primitive refinement laws in a specific order. The following table describes that proof clearly.

*Proof.* The following is the deductive proof of the observer pattern refinement rule.

$$\begin{aligned}
cdecls_1 &= C_1[; pub \cup \{(a, T, v)\}; op \cup \{m_1\{a = le\}\};] \\
&\quad C_2[; pub \cup \{(b, T, v)\}; op;] \\
&\quad C_3[; pub \cup \{(c, T, v)\}; op;] \\
\sqsubseteq &\quad \{\text{Introducing a private attribute and changing a private attribute to public}\} \\
&\quad C_1[; pub \cup \{(a, T, v)\}; op \cup \{m_1\{a = le\}\};] \\
&\quad C_2[; pub \cup \{(b, T, v), (o_1, C_1, \sigma)\}; op;] \\
&\quad C_3[; pub \cup \{(c, T, v), (o_1, C_1, \sigma)\}; op;] \\
\sqsubseteq &\quad \{\text{Adding new method}\} \\
&\quad C_1[; pub \cup \{(a, T, v)\}; op \cup \{m_1\{a = le\}\};] \\
&\quad C_2[; pub \cup \{(b, T, v), (o_1, C_1, \sigma)\}; op \cup \{m_3\{b = o.a\}\};] \\
&\quad C_3[; pub \cup \{(c, T, v), (o_1, C_1, \sigma)\}; op \cup \{m_3\{c = o.a\}\};] \\
\sqsubseteq &\quad \{\text{Extract superclass}\} \\
&\quad S_2[; sig \cup \{\zeta(m_3)\};] \\
&\quad C_1[; pub \cup \{(a, T, v)\}; op \cup \{m_1\{a = le\}\};] \\
&\quad C_2[S_2; pub \cup \{(b, T, v), (o_1, C_1, \sigma)\}; op \cup \{m_3\{b = o.a\}\};] \\
&\quad C_3[S_2; pub \cup \{(c, T, v), (o_1, C_1, \sigma)\}; op \cup \{m_3\{c = o.a\}\};] \\
\sqsubseteq &\quad \{\text{Introduce a superclass}\} \\
&\quad S_1[; pub \cup s = \{(o_2, S_1, (o_2, C_2, \sigma)), (o_3, S_1, (o_3, C_3, \sigma))\}; op \cup \{m_2\{\forall o \in s, o.m_3\}\};] \\
&\quad S_2[; sig \cup \{\zeta(m_3)\};] \\
&\quad C_1[S_1; pub \cup \{(a, T, v)\}; op \cup \{m_1\{a = le\}\};] \\
&\quad C_2[S_2; pub \cup \{(b, T, v), (o_1, C_1, \sigma)\}; op \cup \{m_3\{b = o.a\}\};] \\
&\quad C_3[S_2; pub \cup \{(c, T, v), (o_1, C_1, \sigma)\}; op \cup \{m_3\{c = o.a\}\};] \\
\sqsubseteq &\quad \{\text{Expert pattern for responsibility assignment}\} \\
&\quad S_1[; pub \cup s = \{(o_2, S_1, (o_2, C_2, \sigma)), (o_3, S_1, (o_3, C_3, \sigma))\}; op \cup \{m_2\{\forall o \in s, o.m_3\}\};] \\
&\quad S_2[; sig \cup \{\zeta(m_3)\};] \\
&\quad C_1[S_1; pub \cup \{(a, T, v)\}; op \cup \{m_1\{a = le; m_2\}\};] \\
&\quad C_2[S_2; pub \cup \{(b, T, v), (o_1, C_1, \sigma)\}; op \cup \{m_3\{b = o.a\}\};] \\
&\quad C_3[S_2; pub \cup \{(c, T, v), (o_1, C_1, \sigma)\}; op \cup \{m_3\{c = o.a\}\};] \\
= &\quad cdecls_2
\end{aligned}$$

#### 4.4 Decorator Pattern

The decorator pattern is used to add dynamically new functionalities to an object without altering the original design. The decorator pattern applicability is feasible when we need to add new functionalities composed to the original one when the original behavior should still be available and the execution of the correct behavior can be specified as a conditional statement. The following table describes formally the decorator patterns following to our definition.

*Proof.* The proof of the decorator pattern refinement rule is based on the application of the strategy pattern refinement rule first then inline method, introduce superclass, replace method are presented in [14] the others are presented in [15]

$cdecls_1$	$= C_1[; op \cup \{m_1\{c_A\}\};]$
$p$	$= body(m_1) = \{((c_A; c_B) \triangleleft b_2 \triangleright (c_A; c_C)) \triangleleft b_1 \triangleright c_A\}$
$cdecls_2$	$= S_1[; sig \cup \{\varsigma(m_1)\}]$ $C_1[; pub \cup \{(a, S_1, null)\}; op \cup \{m_1\{(C_{21}.new(a)[\forall x \in \bar{x}_1 \cup \bar{x}_2, a.x = self.x] \triangleleft$ $b_2 \triangleright C_{22}.new(a)[\forall x \in \bar{x}_1 \cup \bar{x}_3, a.x = self.x] \triangleleft b_1 \triangleright C_{11}.new(a)[\forall x \in \bar{x}_1, a.x = self.x]; a.m_1\}\};]$ $C_{11}[S_1; op \cup \{m_1\{c_A\}\};]$ $C_{12}[S_1; pub \cup \bar{x}_1 \cup \{(a, S_1, (a, C_{11}, \sigma))\}; op \cup \{m_1\{a.m_1\}\};]$ $C_{21}[C_{12}; pub \cup \bar{x}_1 \cup \bar{x}_2; op \cup \{m_1\{super.m_1; m_2\}, m_2\{c_B\}\};]$ $C_{22}[C_{12}; pub \cup \bar{x}_1 \cup \bar{x}_3; op \cup \{m_1\{super.m_1; m_3\}, m_3\{c_C\}\};]$

Table 4: The Decorator Pattern Specification

with method call and replace method with method object, in such order. The following is the detailed description of the refinement rule associated to the decorator pattern.

$$\begin{aligned}
cdecls_1 &= C_1[; op \cup \{m_1\{((c_A; c_B) \triangleleft b_2 \triangleright (c_A; c_C)) \triangleleft b_1 \triangleright c_A\}\};] \\
&\sqsubseteq \{\text{Strategy Pattern}\} \\
&S_1[; sig \cup \{\zeta(m_1)\}] \\
&C_1[; pub \cup \{(a, S_1, null)\}; op \cup \{m_1\{(C_{21}.new(a)[\forall x \in \bar{x}_1 \cup \bar{x}_2, a.x = self.x] \triangleleft b_2 \triangleright \\
&\quad C_{22}.new(a)[\forall x \in \bar{x}_1 \cup \bar{x}_3, a.x = self.x] \triangleleft b_1 \triangleright C_{11}.new(a)[\forall x \in \bar{x}_1, a.x = self.x]; a.m_1\}\}\};] \\
&C_{11}[S_1; pub \cup \bar{x}_1; op \cup \{m_1\{c_A\}\};] \\
&C_{21}[S_1; pub \cup \bar{x}_1 \cup \bar{x}_2; op \cup \{m_1\{c_A; c_B\}\};] \\
&C_{22}[S_1; pub \cup \bar{x}_1 \cup \bar{x}_3; op \cup \{m_1\{c_A; c_C\}\};] \\
&\sqsubseteq \{\text{Extract method for } C_{21}.m_1 \text{ and } C_{22}.m_1\} \\
&S_1[; sig \cup \{\zeta(m_1)\}] \\
&C_1[; pub \cup \{(a, S_1, null)\}; \\
&\quad op \cup \{m_1\{(C_{21}.new(a) \triangleleft b_2 \triangleright C_{22}.new(a)) \triangleleft b_1 \triangleright C_{11}.new(a); a.m_1\}\}\};] \\
&C_{11}[S_1; pub \cup \bar{x}_1; op \cup \{m_1\{c_A\}\}\};] \\
&C_{21}[S_1; pub \cup \bar{x}_1 \cup \bar{x}_2; op \cup \{m_1\{c_A; m_2\}, m_2\{c_B\}\}\};] \\
&C_{22}[S_1; pub \cup \bar{x}_1 \cup \bar{x}_3; op \cup \{m_1\{c_A; m_3\}, m_3\{c_C\}\}\};] \\
&\sqsubseteq \{\text{Extract superclass}\} \\
&S_1[; sig \cup \{\zeta(m_1)\}] \\
&C_1[; pub \cup \{(a, S_1, null)\}; \\
&\quad op \cup \{m_1\{(C_{21}.new(a) \triangleleft b_2 \triangleright C_{22}.new(a)) \triangleleft b_1 \triangleright C_{11}.new(a); a.m_1\}\}\}\};] \\
&C_{11}[S_1; pub \cup \bar{x}_1; op \cup \{m_1\{c_A\}\}\};] \\
&C_{12}[S_1; pub \cup \bar{x}_1; op \cup \{m_1\{c_A\}\}\};] \\
&C_{21}[C_{12}; pub \cup \bar{x}_1 \cup \bar{x}_2; op \cup \{m_1\{super.m_1; m_2\}, m_2\{c_B\}\}\};] \\
&C_{22}[C_{12}; pub \cup \bar{x}_1 \cup \bar{x}_3; op \cup \{m_1\{super.m_1; m_3\}, m_3\{c_C\}\}\};] \\
&\sqsubseteq \{\text{Replace method with method object}\} \\
&S_1[; sig \cup \{\zeta(m_1)\}] \\
&C_1[; pub \cup \{(a, S_1, null)\}; \\
&\quad op \cup \{m_1\{(C_{21}.new(a) \triangleleft b_2 \triangleright C_{22}.new(a)) \triangleleft b_1 \triangleright C_{11}.new(a); a.m_1\}\}\}\};] \\
&C_{11}[S_1; pub \cup \bar{x}_1; op \cup \{m_1\{c_A\}\}\};] \\
&C_{12}[S_1; pub \cup \bar{x}_1 \cup \{(a, S_1, (a, C_{11}, \sigma))\}; op \cup \{m_1\{a.m_1\}\}\};] \\
&C_{21}[C_{12}; pub \cup \bar{x}_1 \cup \bar{x}_2; op \cup \{m_1\{super.m_1; m_2\}, m_2\{c_B\}\}\}\};] \\
&C_{22}[C_{12}; pub \cup \bar{x}_1 \cup \bar{x}_3; op \cup \{m_1\{super.m_1; m_3\}, m_3\{c_C\}\}\}\};] \\
&= cdecls_2
\end{aligned}$$

□

## 4.5 Composite Pattern

The composite pattern aim is to enhance a design by presenting a hierarchy of classes by composing objects. The composite pattern is used for a design that contains two kinds of object classes: composite classes, who contain at least one reference object to a class such that one an object of this class is created another object of the referenced objects and a functionality of an object of the composite class is redirected to the object references; and leaf classes who do not contain any object doesn't satisfy the above propriety.

*Proof.* The following is the deductive proof of the composite pattern refinement rule.

$$\begin{aligned}
cdecls_1 &= C_1[; pub; op;] \\
&C_2[; op \cup \{m\};] \\
&\sqsubseteq \{\text{Introducing private attributes and changing private attributes to public}\} \\
&C_1[; pub \cup s = \{(a, S_1, (a, C, \sigma)) \mid C \in cname(cdecls_1)\}; op;]
\end{aligned}$$

$cdecls_1$	$= C_1[; pub; op; ]$ $C_2[; op \cup \{m\}; ]$
$p$	$= C_1 \models p'(C_1, C_2) \wedge C_2 \not\models p'(C_2, C_1) \mid$ $p'(C_i, C_j) = \forall o_1 \in \Pi(C_i), \exists s \subseteq \Pi(C_i) \cup \Pi(C_j), \exists m \in op(C_i) : o_1.m \Rightarrow \forall o_2 \in s, o_2.m$
$cdecls_2$	$= S_1[; sig \cup \{\varsigma(m)\}; ]$ $C_1[S_1; pub \cup s = \{(a, S_1, (a, C, \sigma)) \mid C \in cname(cdecls_1)\}; op \cup \{m\{\forall a \in s, a.m\}\}$ $C_2[S_1; op \cup \{m\}; ]$

Table 5: The Composite Pattern Specification

$$\begin{aligned}
& C_2[; op \cup \{m\}; ] \\
\sqsubseteq & \text{\{Add new method\}} \\
& C_1[; pub \cup s = \{(a, S_1, (a, C, \sigma)) \mid C \in cname(cdecls_1)\}; op \cup \{m\{\forall a \in s, a.m\}\}; ] \\
& C_2[; op \cup \{m\}; ] \\
\sqsubseteq & \text{\{Extract a superclass\}} \\
& S_1[; sig \cup \{\varsigma(m)\}; ] \\
& C_1[S_1; pub \cup s = \{(a, S_1, (a, C, \sigma)) \mid C \in cname(cdecls_1)\}; op \cup \{m\{\forall a \in s, a.m\}\}] \\
& C_2[S_1; op \cup \{m\}; ] \\
= & cdecls_2
\end{aligned}$$

□

## 4.6 Abstract Factory Pattern

The abstract factory pattern is used in the case when a collection of related objects must be created together to satisfy a system requirement, and each object has more than one behavior depends on a situation. It allows the system to get always the correct behavior of the object collection for each situation. This context can be formalized following our model as follows:

$cdecls_1$	$= C_1[; op \cup \{m_1\{(c_{11}; c_{12}) \triangleleft b \triangleright (c_{21}; c_{22})\}\}; ]$
$p$	$= \exists C[; op \cup \{m_1\}; ], C'[; op \cup \{m_2\}; ],$ $C_{11}[; op \cup \{m\{c_{11}\}\}; ], C_{12}[; op \cup \{m\{c_{12}\}\}; ], C_{21}[; op \cup \{m\{c_{21}\}\}; ],$ $C_{22}[; op \cup \{m\{c_{22}\}\}; ] \in cdecls : body(C_1.m_1) \sqsubseteq \{C.new(a_1) \triangleleft b \triangleright C'.new(a_2); a_1.m_1; a_2.m_2\}$
$cdecls_2$	$= S_1[; sig \cup \{\varsigma(m_1)\}]$ $S_2[; sig \cup \{\varsigma(m_2)\}]$ $S_3[; sig \cup \{\varsigma(m_3)\}]$ $C_{11}[S_1; op \cup \{m_1\{c_{11}\}\}; ]$ $C_{12}[S_1; op \cup \{m_1\{c_{12}\}\}\]; ]$ $C_{21}[S_2; op \cup \{m_2\{c_{21}\}\}\]; ]$ $C_{22}[S_2; op \cup \{m_2\{c_{22}\}\}\]; ]$ $C_{31}[S_3; pub \cup \{(a_1, S_1, null), (a_2, S_2, null)\};$ $op \cup \{m_3\{C_{11}.new(a_1); C_{21}.new(a_2); a_1.m_1; a_2.m_2\}\}\]; ]$ $C_{32}[S_3; pub \cup \{(a_1, S_1, null), (a_2, S_2, null)\};$ $op \cup \{m_3\{C_{12}.new(a_1); C_{22}.new(a_2); a_1.m_1; a_2.m_2\}\}\]; ]$ $C_1[; pub \cup \{(a_1, S_1, null), (a_2, S_2, null)\}; op \cup \{m_1\{(C_{31}.new(a_1) \triangleleft b \triangleright C_{32}.new(a_2))\}\}\]; ]$

Table 6: The Abstract Factory Pattern Specification

*Proof.* The above refinement rule looks hard to understand but it is easy to proof. The above rule can be derived by applying the strategy pattern refinement rule proved in section (4.2.) to  $C_1$  and  $C_2$  of  $cdecls_1$ , respectively. Then, we apply it again to the updated class structure  $C_3$  to get the abstract factory pattern design model.

$$\begin{aligned}
cdecls_1 &= C_1[; pub \cup \bar{x}_b; op \cup \{m_1\{(c_{11}; c_{12}) \triangleleft b \triangleright (c_{21}; c_{22})\}\};] \\
&\sqsubseteq \{(c_{11}; c_{12}) \triangleleft b \triangleright (c_{21}; c_{22}) \Leftrightarrow (c_{11} \triangleleft b \triangleright c_{21}); (c_{12} \triangleleft b \triangleright c_{22})\} \\
&\quad C_1[; pub \cup \bar{x}_b; op \cup \{m_1\{c_{11} \triangleleft b \triangleright c_{21}; c_{12} \triangleleft b \triangleright c_{22}\}\};] \\
&\sqsubseteq \{\text{replace method with method object}\} \\
&\quad C_1[; pub \cup \bar{x}_b; pub \cup \{(a_2, C_2, null), (a_3, C_3, null)\}; \\
&\quad op \cup \{m_1\{C_2.new(a_2)[\forall x \in \bar{x}_b, C_2.x = self.x]; C_3.new(a_3)[\forall x \in \bar{x}_b, C_2.x = self.x]; a_2.m_2; a_3.m_3\}\};] \\
&\quad C_2[; pub \cup \bar{x}_b; op \{m_2\{c_{11} \triangleleft b \triangleright c_{21}\}\};] \\
&\quad C_3[; pub \cup \bar{x}_b; op \{m_3\{c_{12} \triangleleft b \triangleright c_{22}\}\};] \\
&\sqsubseteq \{\text{Strategy Pattern to } C_2 \text{ and } C_3\} \\
&\quad S_1[; sig \cup \{\varsigma(m_1)\}] \\
&\quad S_2[; sig \cup \{\varsigma(m_2)\}] \\
&\quad C_{11}[S_1; pub \cup \bar{x}_{11}; op \cup \{m_1\{c_{11}\}\};] \\
&\quad C_{12}[S_1; pub \cup \bar{x}_{12}; op \cup \{m_1\{c_{12}\}\};] \\
&\quad C_{21}[S_2; pub \cup \bar{x}_{21}; op \cup \{m_2\{c_{21}\}\};] \\
&\quad C_{22}[S_2; pub \cup \bar{x}_{22}; op \cup \{m_2\{c_{22}\}\};] \\
&\quad C_1[; pub \cup \bar{x}_b; pub \cup \{(a_2, C_2, null), (a_3, C_3, null)\}; \\
&\quad op \cup \{m_1\{C_2.new(a_2)[\forall x \in \bar{x}_b, C_2.x = self.x]; C_3.new(a_3)[\forall x \in \bar{x}_b, C_2.x = self.x]; a_2.m_2; a_3.m_3\}\};] \\
&\quad C_2[; pub \cup \bar{x}_1 \cup \{(a, S_1, null)\}; \\
&\quad op \cup \{m_1\{C_{11}.new(a)[\forall x \in \bar{x}_{11}, a.x = self.x] \triangleleft b \triangleright C_{12}.new(a)[\forall x \in \bar{x}_{12}, a.x = self.x]\}\};] \\
&\quad C_3[; pub \cup \bar{x}_2 \cup \{(a, S_2, null)\}; \\
&\quad op \cup \{m_2\{C_{21}.new(a)[\forall x \in \bar{x}_{21}, a.x = self.x] \triangleleft b \triangleright C_{22}.new(a)[\forall x \in \bar{x}_{22}, a.x = self.x]\}\};] \\
&\sqsubseteq \{\text{Inline the classes } C_1, C_2, C_3 \text{ and inline the methods } m_1, m_2, m_3\} \\
&\quad S_1[; sig \cup \{\varsigma(m_1)\}] \\
&\quad S_2[; sig \cup \{\varsigma(m_2)\}] \\
&\quad C_{11}[S_1; pub \cup \bar{x}_{11}; op \cup \{m_1\{c_{11}\}\};] \\
&\quad C_{12}[S_1; pub \cup \bar{x}_{12}; op \cup \{m_1\{c_{12}\}\};] \\
&\quad C_{21}[S_2; pub \cup \bar{x}_{21}; op \cup \{m_2\{c_{21}\}\};] \\
&\quad C_{22}[S_2; pub \cup \bar{x}_{22}; op \cup \{m_2\{c_{22}\}\};] \\
&\quad C_1[; pub \cup \bar{x}_b \cup \bar{x}_1 \cup \bar{x}_2 \cup \{(a_1, S_1, null), (a_2, S_2, null)\}; \\
&\quad op \cup \{m_3\{(C_{11}.new(a_1)[\forall x \in \bar{x}_{11}, a_1.x = self.x]; C_{21}.new(a_2)[\forall x \in \bar{x}_{21}, a_2.x = self.x]) \triangleleft b \triangleright \\
&\quad (C_{12}.new(a_1)[\forall x \in \bar{x}_{12}, a.x = self.x]; C_{22}.new(a_2)[\forall x \in \bar{x}_{22}, a.x = self.x])\}\};] \\
&\sqsubseteq \{\text{Strategy pattern to } C_1\} \\
&\quad S_1[; sig \cup \{\varsigma(m_1)\}] \\
&\quad S_2[; sig \cup \{\varsigma(m_2)\}] \\
&\quad S_3[; sig \cup \{\varsigma(m_3)\}] \\
&\quad C_{11}[S_1; pub \cup \bar{x}_{11}; op \cup \{m_1\{c_{11}\}\};] \\
&\quad C_{12}[S_1; pub \cup \bar{x}_{12}; op \cup \{m_1\{c_{12}\}\};] \\
&\quad C_{21}[S_2; pub \cup \bar{x}_{21}; op \cup \{m_2\{c_{21}\}\};] \\
&\quad C_{22}[S_2; pub \cup \bar{x}_{22}; op \cup \{m_2\{c_{22}\}\};] \\
&\quad C_{31}[S_3; pub \cup \bar{x}_{11} \cup \bar{x}_{21}; \\
&\quad op \cup \{m_3\{C_{11}.new(a_1)[\forall x \in \bar{x}_{11}, a_1.x = self.x]; C_{21}.new(a_2)[\forall x \in \bar{x}_{21}, a_2.x = self.x]\}\};] \\
&\quad C_{32}[S_3; pub \cup \bar{x}_{21} \cup \bar{x}_{22}; \\
&\quad op \cup \{m_3\{C_{21}.new(a_1)[\forall x \in \bar{x}_{12}, a_1.x = self.x]; C_{22}.new(a_2)[\forall x \in \bar{x}_{22}, a_2.x = self.x]\}\};] \\
&\quad C_1[; pub \cup \bar{x}_1 \cup \bar{x}_2 \cup \{(a_1, S_1, null), (a_2, S_2, null)\}; \\
&\quad op \cup \{m_1\{(C_{31}.new(a_1)[\forall x \in \bar{x}_{11} \cup \bar{x}_{21}, a_1.x = self.x] \triangleleft b \triangleright \\
&\quad C_{32}.new(a_2)[\forall x \in \bar{x}_{12} \cup \bar{x}_{22}, a.x = self.x])\}\};] \\
&= cdecls_2
\end{aligned}$$

□

## 4.7 Singleton Pattern

The singleton pattern is used to ensure that a specific class has only one instance during the system execution. The following table describes the context, problem and the solution of the singleton pattern following our model.

$cdecls_1$	$= C_1[; pub \cup \{(a, C_2, null)\}; op \cup \{m_1\{c_A; C_2.new(a)[c]; c_B\}\};]$ $C_2[; pri;]$
$p$	$= \forall o_1, o_2 \in \Pi(C_2), o_1 = o_2$
$cdecls_2$	$= C_1[; pub \cup \{(a, C_2, null)\}; op \cup \{m_1\{c_A; C_2.new(a)[skip \triangleleft b \triangleright (b = true; c)]; c_B\}\};]$ $C_2[; pri \cup \{(b, boolean, false)\};]$

Table 7: The Singleton Pattern Specification

*Proof.* The above refinement rule can be proved by introducing a private boolean attribute and introduce an assertion based on it.

$$\begin{aligned}
cdecls_1 &= C_1[; pub \cup \{(a, C_2, null)\}; op \cup \{m_1\{c_A; C_2.new(a)[c]; c_B\}\};] \\
&\quad C_2[; pri;] \\
&\sqsubseteq \{\text{Introduce private attribute to } C_2\} \\
&\quad C_1[; pub \cup \{(a, C_2, null)\}; op \cup \{m_1\{c_A; C_2.new(a)[c]; c_B\}\};] \\
&\quad C_2[; pri \cup \{(b, boolean, false)\};] \\
&\sqsubseteq \{\text{Introduce assertion}\} \\
&\quad C_1[; pub \cup \{(a, C_2, null)\}; op \cup \{m_1\{c_A; C_2.new(a)[skip \triangleleft b \triangleright (b = true; c)]; c_B\}\};] \\
&\quad C_2[; pri \cup \{(b, boolean, false)\};] \\
&= cdecls_2
\end{aligned}
\quad \square$$

## 5 Assessment and Comparison

The aim of the assessment is to clarify the suitability of each formal model used for the purpose of patterns specification, it also identifies the merits and the demerits of the existing specification and addresses the limitations of the different models. This section introduces an assessment of the formal specifications presented in section 2 compared to our specification.

### 5.1 The Assessment criteria

In fact, the assessment criteria used in this paper are based on what proposed in [2] and [10]. The proposed criteria can be classified into three main aspects:

### 5.1.1 Formalism

The formalism shows the mathematical foundation of each model also shows either the proposed model has a visual notation or not. The formalism proposed by each model provides an indication of the complexity that can be show from the specification of the desired system. For example, the predicate logic is the preferred and the most simple formalism can be used for formal specification. Also any specification similar to a software development language makes the specification more easier.

### 5.1.2 Completeness

The completeness aspect is an important one for the any assessment. In our assessment model, the assessment stands for the ability of the proposed model to specify both the problem and solution of a pattern including the structure and the behavior of the pattern solution. Also, a complete formal specification should address both the instantiation and the composition issues. A specification that is able to specify all these aspect can be evaluated to a complete one.

### 5.1.3 Ease-of-Use

The ease-of-use refers to the size of vocabularies used by each model. A low value of vocabulary is desired for a good specification.

## 5.2 The Assessment results

The following table provides a summary of the assessment criteria applied the existing formal specification of the design patterns including our model.

	Formalism		Completeness				Ease-To-Use	
	Math-Found	Visual	Solution		Problem	Context		Instantiation
			Structure	Behavior				
Bayley	FOL	none	FOL	none	none	none	mapping	> 25 terms
BPSL	FOL+TLA	none	FOL	TLA	none	none	mapping	6 per + ? TRs
ElePUS	HOML+FOL	LePUS3	FOL	FOL	none	none	parameters	36 relations
RSL	VDM+AS	none	FOL	FOL	none	none	parameters	36 relations
UML-B	FOL+WP+Sets	UML	FOL	WP	none	none	UML	8 terms
B	FOL+WP+Sets	none	FOL	WP	none	none	inclusion + renaming	> 8 terms
Disco	TLA	none	Disco Syntax	TLA	none	none	Concrete statistics data	5 terms
Prolog	FOL	none	FOL	FOL	none	none	mapping	7 prim + 3 opers
rCOS	UTP	GTS	rCOS Syntax	UTP Designs	yes	yes(FOL)	mapping	16 terms

Table 8: Evaluation of Formal Pattern Specification Approaches

Form the above table, the particularity of our model is shown clearly through its ability to describe the problem, the solution and the context of patterns and its easiest to use compared to some other proposed models. Furthermore, the proposed model is extensible, in such way some other object oriented concepts can be integrated easily such as the static methods and attributes.

## 6 Conclusion & Future Work

In this report, a model for formalizing design patterns based on rCOS specification is described and called object oriented design model (OODM), seven design patterns (i.e. adapter, strategy, observer, decorator, composite, abstract factory and singleton) have been formalized following that model. The model considers a design pattern as a 3-tuple consists of two OODMs describing the problem and solution designs and a predicate over the variable structures that is not satisfied by the OODM describing the problem and it is satisfied by the OODM describing the solution. Furthermore, the relationship between the problem and the solution designs are specified as refinement relation that has been proved for each pattern using deductive reasoning based on the primitive refinement rules already defined for rCOS. As a future work, a pattern composition through our model will be addressed, a model checker for verifying the correct composition tool support developed.

## References

- [1] Taibi, T., Ling, D.: Formal Specification of Design Patterns-A Balanced Approach. In the Journal of Object Technology, Vol.2, no. 4, pp. 127-140, August, 2003.
- [2] Sivakumar, C.: ELePUS: Extended Language for Pattern Uniform. Msc thesis, Purdue University, August, 2000. Available at <http://www.cs.iupui.edu/~rraje/pub/elepus.pdf>
- [3] Eden, A.H, Hirshfeld, Y.: Principles in Formal Specification of Object Oriented Design and Architecture. In the proceeding of the 2001 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'01), Toronto, Ontario, Canada, IBM Press, November, 2001. Available at <http://www.eden-study.org/articles/2001/cascon.pdf>
- [4] Aranda, G., Moore, R.: Formally Modelling Compound Design Patterns. Technical Report no. 225, UNU-IIST, P.O. Box 3058, Macau, December 2000. Available at <http://www.iist.unu.edu/>
- [5] Bayley, I. Zhu, H.: Formal Specification of Design Patterns as Structural Properties. Technical Report, Department of Computing, School of Technology, Oxford Brookes University, Oxford, UK, January 2007. Available at <http://cms.brookes.ac.uk/staff/HongZhu/Publications/DPTechReport.pdf>

- [6] Amnon, H.E.: LePUS—A Declarative Pattern Specification Language. Technical report no. 326, Department of Computer Science, Tel Aviv University, 1998.
- [7] Gamma, E., Helm, R., Johnson, R., and Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.
- [8] Alan, S., James, R.T.: Design Patterns Explained: A New Perspective on Object-Oriented Design. Software Pattern Series, Addison-Wesley, 2002.
- [9] Cechich, A., Moore, R.: A Formal Basis for Object-Oriented Patterns. In the proceedings of 6th Asia-Pacific Software Engineering Conference (APSEC'99) Takamatsu, Japan, IEEE Computer Society Press, pp. 284-291, December, 1999.
- [10] Blazy, S., Gervais, F., Laleau, R.: Reuse of Specification Patterns with the B Method. In the proceeding of the 3rd International Conference of B and Z Users, Turku, Finland, LNCS, vol. 2651, pp. 40-57, February, 2004.
- [11] Marcano, R., Lévy, N., Losavio, F.: Spécification et Spécialisation de Patterns en UML et B. In the proceeding of Object Models and Languages (LMO'2000), Montreal, Canada, Hermes edition, pp. 245-260, January, 2000.
- [12] Meyer, E., Souquière, J.: A Systematic Approach to Transform OMT Diagrams to a B Specification. In the proceeding World Congress of Formal Methods in the Development of Computing Systems(FM'99), Toulouse, France, LNCS, vol. 1708, pp. 875-895, September, 1999.
- [13] Cavalcanti, A., Sampaio, A., Woodcock, J.: Refinement: An Overview. Chapter 1 of the book of Refinement Techniques in Software Engineering, LNCS, Springer Berlin, Heidelberg, Vol. 3167, pp. 1-17, October, 2006.
- [14] He, J., Li, X., Liu, Z.: rCOS: A Refinement Calculus of Object Systems. Formal Methods for Components and Objects, Theoretical Computer Science, Vol.365, n.1-2, pp.109-142, Elsevier, November, 2006.
- [15] Quan, L., He, J., Liu, Z.: Refactoring and Pattern-directed Refactoring: A Formal Perspective. Technical Report 318, UNU-IIST, P.O.Box 3058, Macau, January 2005. Presented and published in the proceedings of Australian Software Engineering Conference 2005, IEEE Computer Society Press 2005. Available at <http://www.iist.unu.edu/>
- [16] Boas, P. E.: Resistance is Futile; Formal Linguistic Observations on Design Patterns. Research Report CT-1997-02, The Institute For Logic, Language, and Computation (ILLC), University of Amsterdam, February, 1997. Available at <http://www.illc.uva.nl/Publications/ResearchReports/CT-1997-02.text.ps.gz>
- [17] Dong, J., Alencar P., Cowan, D.: Ensuring Structure and Behavior Correctness in Design Composition. In the Proceedings of the 7th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems(ECBS), pages 279-287, Edinburgh UK, April 2000.
- [18] Kerievsky J.: Refactoring To Patterns, Addison-Wesley Professional, August, 2004.

- [19] Fowler, M.: Refactoring: Improving the Design of Existing Code. The Addison-Wesley Object Technology Series. Addison-Wesley, 1999
- [20] Mikkonen, T.: Formalizing Design Patterns. In the proceeding of the 20th IEEE International Conference on Software Engineering (ICSE'98), IEEE Computer Society Press, pp. 115-124, 1998.
- [21] Alencar, P., Cowan, D., Dong, J., Lucena, C.: A Pattern-Based Approach to Structural Design Composition. In the proceeding of the 23rd IEEE Annual International Computer Software and Application Conference, IEEE Computer Society Press, October, 1999.
- [22] Alencar, P., Cowan, D., Lucena, C.: A Formal Approach to Architectural Design Patterns. In the Proceedings of the Third International Symposium of Formal Methods Europe on Industrial Benefit and Advances in Formal Methods (FME'96), London, UK, LNCS, Springer-Verlag, vol. 1051, pp. 576-594, 1996.
- [23] Abadi, M., Cardelli, L. A Theory Of Objects, Springer, 1996.
- [24] Liu, Z., Jifeng, H. Li, X. rCOS: Refinement of Component and object Systems, In the proceeding of the Third International Symposium on Formal Methods for Components and Objects (FMCO'04), Leiden, The Netherlands, Springer-Verlag Berlin Heidelberg, LNCS, vol. 3657, pp. 183-221, November, 2005.
- [25] Hoare, C.A.R., He, J. Unifying Theory of Programming, Prentice-Hall, 1998.
- [26] Cavalcanti, A., Numann, D.A. A Weakest Precondition Semantics for an Object-Oriented Language of Refinement, In the proceeding of Formal Methods (FM'99): World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, vol. II, Springer-Verlag Berlin Heidelberg, LNCS, vol. 1709, pp. 1439-1459, september, 1999.
- [27] Taibi, T. Formalizing Design Patterns Composition, In the IEE-Proceeding Software, vol. 153, n. 3, pp. 127-136, June, 2006.
- [28] Yacoub, M.S, Xue, H., Ammar, H.A. POD: A Composition Environment for Pattern-Oriented Design, In the Proceeding of the 34th International Conference on Technology of Object-Oriented Languages and Systems (TOOL'34), Santa Barbara, CA, USA, IEEE Computer Society, pp. 263-272, July, 2000.
- [29] Yacoub, M.S, Ammar, Ail, M. Constructional Design Patterns as Reusable Components, In the proceedings of the 6th International Conference of Software Reuse: Advances in Software Reusability, (ICSR-6), Vienna, Austria, LNCS, Springer Berlin Heidelberg, vol. 1844, pp. 369-387, June, 2000.
- [30] Yacoub, M.S, Ammar, H.A. Pattern-Oriented Analysis and Design: Composing Patterns to Design Software Systems, Addison Wesley, 2003.
- [31] Christopher A., The Timeless Way of Building, Oxford University Press, 1979.