The United Nations University

UNU/IIST

International Institute for
Software Technology

# GoF Behavioural Patterns: A Formal Specification

Luis Reynoso and Richard Moore

May 2000

UNU/IIST Report No. 201

# 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 Macau, and was founded in 1991. It started operations in July 1992. UNU/IIST is jointly funded by the Governor of Macau and the governments of the People's Republic of China and Portugal through a contribution to the UNU Endownment Fund. As well as providing two-thirds of the endownment fund, the Macau 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. 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 $\boxed{\mathsf{R}}$, Technical $\boxed{\mathsf{T}}$, Compendia $\boxed{\mathsf{C}}$ or Administrative $\boxed{\mathsf{A}}$. 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, Macau or visit UNU/IIST home page: http://www.iist.unu.edu, if you would like to know more about UNU/IIST and its report series.

Zhou Chaochen, Director — 01.8.1997 – 31.7.2001

The United Nations University

UNU/IIST

International Institute for
Software Technology

P.O. Box 3058
Macau

# GoF Behavioural Patterns: A Formal Specification

## Luis Reynoso and Richard Moore

### Abstract

GoF behavioural patterns are widely used in object-oriented design when dealing with communication between or the transfer of responsibilities between classes and objects. The GoF catalogue describes, using a standard but informal notation, eleven behavioural patterns which can be used to capture different aspects of behaviour in a design. In this paper, we present an analysis of the essential components and properties of nine of these patterns, including not only the properties which are described explicitly in the GoF catalogue but also properties which are not stated explicitly but which can be deduced from the description, intent, motivation, class and method names, and so on of the pattern. These properties are then specified formally, using a formal model of a general object-oriented design which was developed in earlier work as the basis for the specification, and we also formally specify how to check whether a subset of a particular design matches a particular pattern.

Luis Reynoso is a Fellow of UNU/IIST (November 1999 to May 2000), on leave from Comahue University, Neuquén, Argentina, where he is teaching assistant. His research interests are focused on the combination of formal and informal methods and software engineering. He also works in the Cadastral Provincial Direction of the Public Administration of the government of the province of Neuquén.

Richard Moore is a Research Fellow on the staff of UNU/IIST, a position he took up on October 1st 1995. He has an M.A. in mathematics from the University of Cambridge and a Ph.D. in physics from the University of Manchester. He has been engaged in computing science research in the field of formal methods since 1985, a large part of which was carried out in the formal methods group at Manchester University. He has written several papers on formal methods and is co-author of two books on formal methods – mural: a Formal Development Support System; and Proof in VDM: A Practitioner's Guide. He has also worked for the Defence Research Agency in Malvern, UK, on various formal methods projects, both as a consultant and as a full-time member of staff.

# Contents

# 1 Introduction

Design patterns are the product of one cognitive intellectual activity, abstraction, “a fundamental objective of good software development” [3]. The patterns are generic and embody “best practice” solutions to a particular range of design problems, though these solutions are not necessarily the simplest or most efficient for any given problem [6]. Patterns thus offer designers a way of reusing proven solutions to particular aspects of design rather than having to start each new design from scratch.

Design patterns are also useful because they provide designers with an effective “shorthand” for communicating with each other about complex concepts [1]: the name of the pattern serves as a precise and concise way of referring to a design technique which is well-documented and which is known to work well.

One specific and popular set of software design patterns, which are independent of any specific application domain, are the so-called “GoF”[1] patterns which are described in the catalogue of Gamma et al. [5]. The GoF catalogue is thus a description of the know-how of expert designers in problems appearing in various different domains.

Although there is nothing in design patterns that makes them inherently object-oriented [1], the GoF catalogue uses object-oriented concepts to describe twenty three patterns which capture and compact the essential parts of corresponding design solutions. Each GoF pattern thus identifies a group of classes, together with the key aspects of their functionality and interactions, which commonly occur in a range of different object-oriented design problems.

In earlier work [4], we have defined an abstract model of a general object-oriented design in terms of classes, their properties, and the relationships between them, and we have formally specified this model using the RAISE specification language RSL [9]. We have also identified common properties of the classes and relationships appearing in the patterns in the GoF catalogue and we have defined these as generic RSL functions. These functions thus offer a means of checking whether (a subset of) a particular design matches a given GoF pattern.

Broadly speaking, this process involves associating various elements of the design (classes, state variables and methods, including their input parameters and results) with the names appearing in the pattern using a *renaming map*. The renaming map thus defines which entity in the design corresponds to which entity in the pattern, or which *role* in the pattern is played by a particular class in the design, and we can then check that each entity appearing in the renaming map at the design level satisfies the properties of the pattern level entity to which it renames, and hence that a (subset of a) design matches a pattern as a whole. This process is illustrated in Figure 1.

In this report we use these techniques to specify the properties of nine of the eleven patterns belonging to the behavioural group of patterns in the GoF catalogue, one of the three groups into which the catalogue is divided. Section 2 analyses the essential components and properties

[1] “Gang of Four”

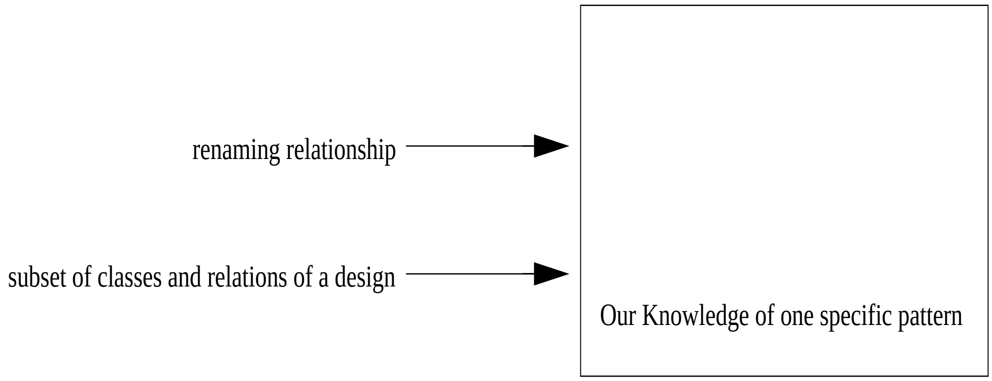


Figure 1: The Elements of the Matching Process

of each of these patterns, taking into account not only properties which are stated explicitly in the GoF catalogue but also properties which are implicit in the description, intent, motivation, class and method names and so on of the pattern. These implicit properties are then used to formulate an extension to the presentation of the pattern's structure, where appropriate. We also give a formal specification of the properties of each pattern in RSL. Note, however, that these specifications rely almost entirely on the specifications of the common properties of patterns presented in [4] which are not repeated here. In addition, familiarity with the elements of the basic model presented in [4] is also assumed when presenting these specifications. In Section 3 we then illustrate how the model can be used to check that an actual design corresponds to a pattern, using the example which appears in the motivation of the State pattern in [5]. Section 4 then uses the analysis from Section 2 to compare and relate the patterns, both conceptually and in terms of their specifications. Finally, in Section 5 we give a brief summary of our work and discuss some possible extensions to it.

## 2 Behavioural Patterns

The GoF patterns have been fundamental in helping the software engineering community recognise forms of evolution for objects in an application domain. Behavioural patterns basically deal with communication between or the transfer of responsibilities between classes and objects, and each pattern identifies an aspect of a system that may vary and proposes a way of writing programs such that the variation is possible [11].

We begin by briefly describing each pattern, stressing the particular aspect of a system that varies with each pattern.

**Chain of Responsibility**
"Potential receivers" serve as the channel of communication between the object that makes a request and the actual receiver of a message.

**Command**

Command objects define operations on components and other objects. Undoable operations can be carried out by command objects. Command allows classes and objects to be used for non-physical things, operations and abstract notations [3].

**Interpreter**

Interpreter represents a grammar as a class hierarchy and implements an interpreter as an operation on instances of these classes [5].

**Iterator**

The Iterator pattern abstracts behaviour for accessing and traversing objects in an aggregate. It is also used to allow the elements of an aggregate structure to be accessed without showing the inner structure (implementation) of the aggregate.

**Mediator**

The Mediator pattern promotes object interaction to full object status. It fosters loose coupling by keeping objects from referring to each other explicitly [12]. Instead of colleagues referring to each other explicitly, each refers exclusively to the mediator so that the mediator acts as the channel of communication between multiple peer objects.

**Memento**

One object coordinates and safekeeps the state of another object, allowing it to recover its previous state. The snapshot of the object is saved in a memento object. The three objects interact with each other in terms of coupled methods that save and recover the class state.

**Observer**

The one-many relationship between objects is defined in this pattern. An "interface for signalling changes in the subjects observed" is implemented by the collaborations of the pattern.

**State**

The State pattern encapsulates the states of an object so that it can change its behaviour when its state changes [5]. This pattern allows an object to appear to change its class.

**Strategy**

The Strategy pattern encapsulates a number of algorithms and lets us vary the algorithm used at run-time [8].

**Template Method**

This pattern is classified by [3] as a cliche, a trite of programming. The rule of thumb underlying this method is: "if a method calls other methods, then by overriding these methods the calling method changes its behaviour".

**Visitor**

Visitor encapsulates behaviour that would otherwise be distributed across classes. It allows behaviour to be added to a composite structure without changing the existing class definitions [10].

These properties are summarised in Table 1.

| Pattern | Aspect that can vary |
|---|---|
| Chain of Responsibility | the object that fulfills a request |
| Command | when and how a request is fulfilled |
| Interpreter | grammar and interpretation of a language |
| Iterator | how elements of an aggregate object are accessed or traversed |
| Mediator | how and which objects interact with each other |
| Memento | what private information is stored outside an object, and when |
| Observer | dependencies between objects |
| State | the behaviour of an object, depending on its state |
| Strategy | an algorithm |
| Template Method | steps of an algorithm |
| Visitor | operations that can be applied to objects |

Table 1: Aspects of Behavioural Patterns

Each of the subsections below deals with a single pattern. We first present the most important elements of each pattern (the intent, structure, participants, and collaborations) in the format used in [5], which has now become the effective standard notation. Then we analyse the properties of the pattern informally, describing not only those properties which are stated explicitly in [5] but also those which are implicit in the pattern's structure, collaborations, intent, motivation, class and method names, and so on. Finally, we present a formal specification of these properties.

Each specification begins with a definition of a set of constants representing the names of the entities (classes, state variables, methods and parameters) appearing in the pattern. Then the properties of the pattern are defined by using these constants as parameters to various functions which describe general properties of patterns. These functions are all defined in [4].

The order in which the patterns are described below is not significant, except that they are generally presented in order of increasing complexity. Particular concepts which apply to several patterns are explained in detail where they first crop up, but only superficially where they reoccur.

## 2.1 Mediator Pattern

The Mediator pattern is used "when a set of objects communicate in well-defined but complex ways so that the resulting interdependencies are unstructured and difficult to understand; when reusing an object is difficult because it refers to and communicates with many other objects;

and when a behaviour that's distributed between several classes should be customizable without a lot of subclassing" [5]. It is one of the simplest patterns: its structure, which is shown in Figure 2, includes neither methods nor annotations. It is therefore an appropriate pattern with which to introduce the simple concepts and properties used in the formal specifications.

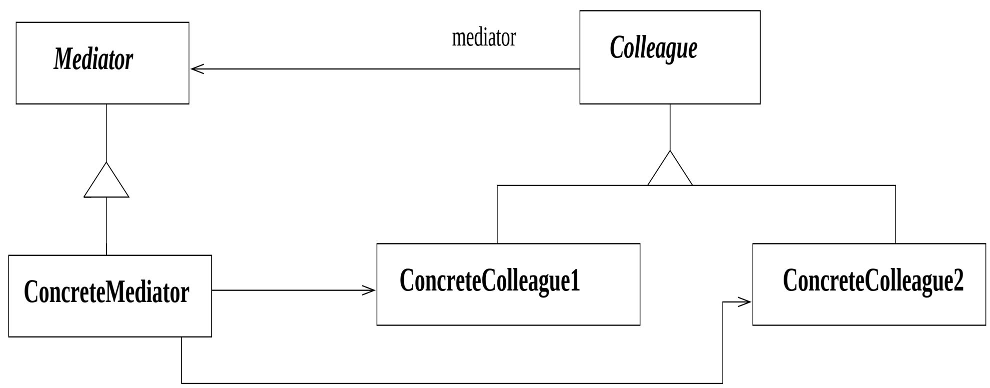


Figure 2: Mediator Pattern Structure

The intent, participants and collaborations of the pattern as defined in [5] are:

**Intent**

Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

**Participants**

**Mediator**

- defines an interface for communicating with Colleague objects.

**ConcreteMediator**

- implements cooperative behaviour by coordinating Colleague objects.
- knows and maintains its colleagues.

**Colleague classes**

- each Colleague class knows its Mediator object.
- each colleague communicates with its mediator whenever it would have otherwise communicated with another colleague.

**Collaborations**

- Colleagues send and receive requests from a Mediator object. The mediator implements the cooperative behaviour by routing requests between the appropriate colleague(s).

As can be seen from its structure in Figure 2, the Mediator pattern consists essentially of two linked hierarchies of classes and their subclasses. Such hierarchies are in fact common in many patterns and share common properties, though the ones appearing in this Mediator pattern are the simplest form. We explain them using the Mediator-ConcreteMediator hierarchy as example.

First, there should be only a single class in the design which plays the Mediator role, and this should be an abstract class. Second, although there may in practice be many levels of classes between the Mediator class and the ConcreteMediator classes, all the *leaf* classes in the hierarchy (that is the classes which are subclasses of the Mediator class but which have no subclasses themselves) must play the ConcreteMediator role. Finally, neither the Mediator class nor any of its subclasses can play a role from outside the hierarchy (i.e. no class in the Mediator hierarchy can play either the Colleague or the ConcreteColleague role), though subclasses of the Mediator class which are not leaf classes may also play the ConcreteMediator role (for example, one concrete mediator may be a specialisation of another).

Such a hierarchy could of course be degenerate – there may be only a single class playing the ConcreteMediator role. In our specification of the pattern, and indeed of hierarchies of classes in all the patterns we deal with, we still consider this case to be a valid instantiation of the pattern. However, there is in practice little to be gained by separating the Mediator and ConcreteMediator roles in the design if there is only a single ConcreteMediator class, and we could combine them into a single role, which would of course be the ConcreteMediator role, and omit the Mediator role from the pattern entirely – indeed it is stated explicitly in the discussion of the implementation of the pattern in [5] that "there is no need to define an abstract Mediator class when colleagues work with only one mediator". We have neglected this aspect in our specification, instead choosing to consider this situation as a *variation* of the pattern. In fact, similar variations can be made to many other patterns and indeed the concept of variations of a pattern is used by many authors when they want to express a refinement or an extension of a pattern. These variations will be the subject of future work.

Another possibility is that a design may contain more than one Mediator hierarchy or more than one Colleague hierarchy, either coupled in pairs as depicted in the Mediator pattern structure or perhaps with some sort of sharing between several mediator and colleague hierarchies. We do not model this situation explicitly, preferring instead to consider it, without any loss of generality, as multiple instances of a Mediator pattern in which both hierarchies are unique – we simply construct a different renaming map for each separate instance of the pattern in the design.

According to the participants and collaborations of the pattern, the ConcreteMediator classes are responsible for implementing cooperative behaviour between colleagues by routing requests between them. For this to make sense, a ConcreteMediator class should have at least two associated ConcreteColleague classes in the design otherwise there is little point in having the mediator – the colleague class is simply communicating with itself. Furthermore, a ConcreteColleague class which is not associated with any ConcreteMediator class can hardly be said to play the ConcreteColleague role since it cannot participate in any communications with other ConcreteColleague classes except directly. Such direct communication between concrete colleagues is considered to be contrary to the spirit of the pattern, which requires that the mediators handle the communication between the colleagues, and we therefore rule it out and require that every ConcreteColleague class is associated with at least one ConcreteMediator class.

Based on these considerations, we can now go on to develop the formal specification of the Mediator pattern.

### 2.1.1 Formal Specification of the Mediator Pattern

We first define the names of the classes, state variables, methods and parameters used in the pattern as RSL constants. For the Mediator pattern, these are specified as follows:

**value**
Mediator : G.Class_Name,
ConcreteMediator : G.Class_Name,
Colleague : G.Class_Name,
ConcreteColleague : G.Class_Name,
mediator : G.Variable_Name

In order to verify whether some given subset of the classes and relationships of a design effectively represent a Mediator pattern, we link the classes in the design to classes in the pattern using the renaming map. The classes in the pattern to which a particular class in the design are linked in this way are called its *roles*[2]. Then, a given subset of the classes and relationships in the design can be considered as a Mediator pattern if each of these classes has properties which are consistent with those of its designated role(s) in the pattern.

These properties are defined by instantiating the general functions defined in [4] with parameters representing the particular names appearing in the Mediator pattern. The resulting specification then represents the specification of the Mediator pattern.

The specific properties of the classes in the Mediator pattern are as follows:

1. there is a single class which plays the Mediator role. This class is abstract and forms the root of a hierarchy of classes in which all leaf classes play the ConcreteMediator role and in which no class plays either the Colleague role or the ConcreteColleague role;

2. there is a single class which plays the Colleague role. This class is abstract and forms the root of a hierarchy of classes in which all leaf classes play the ConcreteColleague role and in which no class plays either the Mediator role or the ConcreteMediator role;

[2] In most cases a class in the design plays a single role (i.e. corresponds to or renames to a single class in the pattern), but it is possible for one class to play several roles.

3. the class which plays the Colleague role contains a single state variable which plays the role of the mediator state variable;

4. the class playing the Colleague role is linked to the class playing the Mediator role by a one-one association relation representing the state variable playing the mediator role, and there are no other relations between these two classes;

5. there is at least one class playing the ConcreteMediator role, and every class playing this role is a concrete subclass of the class which plays the Mediator role;

6. there is at least one class playing the ConcreteColleague role and every class playing this role is a concrete subclass of the class which plays the Colleague role;

7. every class playing the ConcreteMediator role is linked to at least two classes playing the ConcreteColleague role by association or aggregation relations;

8. there are no direct relationships between distinct classes playing the ConcreteColleague role;

9. for every class playing the ConcreteColleague role there is at least one class playing the ConcreteMediator role which is linked to it by an association or aggregation relation.

We use the functions *hierarchy* and *is_abstract_class* from [4] to define the first of these properties. The function *hierarchy* is a generic function which checks that a hierarchy of classes in the design has as its root a class which plays a given role in the pattern and which is unique in the design, has leaf classes which play any of a given set of roles in the pattern (In this case there is only one class in this set, namely ConcreteMediator, but it is possible to have more than one class as, for example, in the Command pattern described in Section 2.8.), and has no classes which play roles from a given set of roles (in this case Colleague and ConcreteColleague). The property we require is then obtained by simply instantiating this function with the required roles.

The generic function *is_abstract_class* checks that all classes that play a given role in the design are abstract, and again we instantiate this with the appropriate role, namely Mediator, to obtain the property we require. The first property of the Mediator pattern is thus specified as follows:

```
value
   Mediator_hierarchy : Wf_Design_Renaming → Bool
   Mediator_hierarchy(dr) ≡
      hierarchy
         (
            Mediator, {ConcreteMediator}, {Colleague, ConcreteColleague}, dr
         ),

   is_abstract_Mediator : Wf_Design_Renaming → Bool
   is_abstract_Mediator(dr) ≡ is_abstract_class(Mediator, dr)
```

The same functions *hierarchy* and *is_abstract_class* are of course used to specify the second property of the Mediator pattern, except that in this case we instantiate the functions with the Colleague and ConcreteColleague roles in place of the Mediator and ConcreteMediator roles and vice versa.

**value**
  Colleague_hierarchy : Wf_Design_Renaming → **Bool**
  Colleague_hierarchy(dr) ≡
    hierarchy
      (
        Colleague, {ConcreteColleague}, {Mediator, ConcreteMediator}, dr
      ),

  is_abstract_Colleague : Wf_Design_Renaming → **Bool**
  is_abstract_Colleague(dr) ≡ is_abstract_class(Colleague, dr)

The third property, that the Colleague class contains a single state variable playing the mediator role, is obtained by similarly instantiating the function *store_unique_variable* with both the Colleague and mediator roles. This is combined in our specification with the specification of the fourth property which defines the relationship between the Colleague and Mediator classes. The function *has_assoc_aggr_var_ren* checks that the two given classes are linked by at least one association or aggregation relation which has given sink cardinality and which represents the given state variable. This is instantiated with the appropriate roles in the function *Mediator_relation* to give the property that there is at least one association relation linking the Colleague and Mediator classes which represents the mediator state variable and which has sink cardinality one. The function *has_unique_assoc_aggr* checks that the two given classes are linked by a unique association or aggregation and that there is no instantiation relation between them. These two constraints together ensure that property 4 is satisfied.

**value**
  Mediator_relation : Wf_Design_Renaming → **Bool**
  Mediator_relation(dr) ≡
    has_assoc_aggr_var_ren
      (Colleague, Mediator, Association, mediator, G.one, dr) ∧
    has_unique_assoc_aggr(Colleague, Mediator, dr) ∧
    store_unique_vble(Colleague, mediator, dr)

The fifth and sixth properties of the Mediator pattern are specified using the functions *exists_role* and *is_concrete*. The first of these checks that there is at least one class in the design that plays a given role, while the second checks that the classes of one given role are concrete with respect to those of another, that is that they are subclasses and none of the methods that are visible in the classes (including inherited methods) are defined (i.e. all visible methods are either implemented or error methods). Note that the subclasses are not necessarily immediate subclasses: they could in fact be separated from the parent class by a hierarchy of intermediate classes.

**value**
  exists_concrete_Mediator : Wf_Design_Renaming → **Bool**
  exists_concrete_Mediator(dr) ≡ exists_role(ConcreteMediator, dr),

  is_concrete_Mediator : Wf_Design_Renaming → **Bool**
  is_concrete_Mediator(dr) ≡
    is_concrete(Mediator, ConcreteMediator, dr),

  exists_concrete_Colleague : Wf_Design_Renaming → **Bool**
  exists_concrete_Colleague(dr) ≡
    exists_role(ConcreteColleague, dr),

  is_concrete_Colleague : Wf_Design_Renaming → **Bool**
  is_concrete_Colleague(dr) ≡
    is_concrete(Colleague, ConcreteColleague, dr)

The functions *has_at_least_two_assoc_aggr*, *classes_not_related* and *has_assoc_aggr* are used to specify the last three properties of the Mediator pattern respectively. With the appropriate parameters, these functions correspond precisely to the properties required.

**value**
  Colleagues_relation : Wf_Design_Renaming → **Bool**
  Colleagues_relation(dr) ≡
    has_at_least_two_assoc_aggr
      (ConcreteMediator, ConcreteColleague, G.one, dr),

  Colleagues_not_related : Wf_Design_Renaming → **Bool**
  Colleagues_not_related(dr) ≡
    classes_not_related(ConcreteColleague, dr),

  Concrete_Colleagues_relation : Wf_Design_Renaming → **Bool**
  Concrete_Colleagues_relation(dr) ≡
    has_assoc_aggr(ConcreteColleague, ConcreteMediator, G.one, dr)

Finally we can define a single function which checks all the required properties and which thus verifies if a design represents a Mediator pattern.

**value**

$$
\begin{array}{l}
\text{is\_mediator} : \text{Wf\_Design\_Renaming} \to \mathbf{Bool} \\
\text{is\_mediator}(\text{dr}) \equiv \\
\quad \text{Mediator\_hierarchy}(\text{dr}) \wedge \\
\quad \text{exists\_concrete\_Mediator}(\text{dr}) \wedge \\
\quad \text{Colleague\_hierarchy}(\text{dr}) \wedge \\
\quad \text{exists\_concrete\_Colleague}(\text{dr}) \wedge \\
\quad \text{Concrete\_Colleagues\_relation}(\text{dr}) \wedge \\
\quad \text{Mediator\_relation}(\text{dr}) \wedge \\
\quad \text{Colleagues\_relation}(\text{dr}) \wedge \\
\quad \text{is\_concrete\_Mediator}(\text{dr}) \wedge \\
\quad \text{is\_abstract\_Mediator}(\text{dr}) \wedge \\
\quad \text{is\_concrete\_Colleague}(\text{dr}) \wedge \\
\quad \text{is\_abstract\_Colleague}(\text{dr}) \wedge \text{Colleagues\_not\_related}(\text{dr})
\end{array}
$$

## 2.2 Template Method Pattern

The Template Method pattern is used "to implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behaviour that can vary; when common behaviour among subclasses should be factored and localised in a common class to avoid code duplication; and to control subclass extensions by defining a method that calls "hook" operations at specific points, thereby permitting extensions only at those points" [5]. The structure of the pattern is shown in Figure 3.

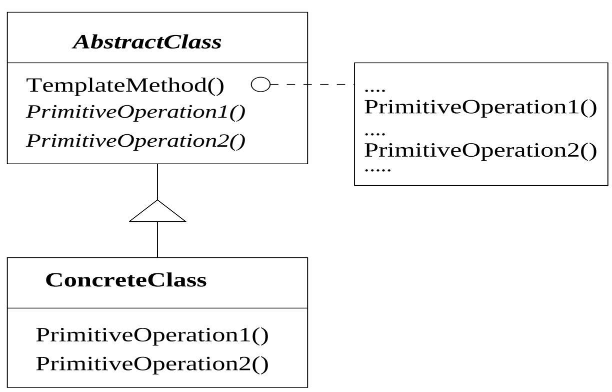


Figure 3: Template Method Pattern Structure

The intent, participants and collaborations of the pattern are defined in [5] as follows:

### Intent

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

**Participants**

**AbstractClass**

- defines abstract primitive operations that concrete subclasses define to implement steps of an algorithm.
- implements a template method defining the skeleton of an algorithm. The template method calls primitive operations as well as operations defined in AbstractClass or those of other objects.

**ConcreteClass**

- implements the primitive operations to carry out subclass-specific steps of the algorithm.

**Collaborations**

- ConcreteClass relies on AbstractClass to implement the invariant steps of the algorithm.

The pattern structure shown in Figure 3 consists of a single inheritance hierarchy in which an abstract class implements a method in terms of operations which are defined locally but implemented in subclasses. The properties of this hierarchy are similar to the properties of the two hierarchies in the Mediator pattern (see Section 2.1) though even simpler – the only classes in the Template Method pattern are AbstractClass and ConcreteClass so we do not need to specifically exclude other pattern classes from the hierarchy and we simply require that all leaf classes in the hierarchy play the ConcreteClass role.

The pattern structure also shows only a single TemplateMethod method in the AbstractClass. We could of course have more than one such method in a design. However, without loss of generality we can consider that case to correspond to multiple instances of the Template Method pattern, one for each method. We therefore restrict here to a single such method. Of course a single TemplateMethod can depend on more than one PrimitiveOperation and these can be implemented at different levels within the hierarchy.

### 2.2.1 Formal Specification of the Template Method Pattern

The names of the classes and methods used in the Template Method pattern are defined as the following RSL constants:

**value**
AbstractClass : G.Class_Name,
ConcreteClass : G.Class_Name,
TemplateMethod : G.Method_Name,
PrimitiveOperation : G.Method_Name

The specific properties of the two classes in the Template Method pattern are as follows:

1. there is a single class which plays the AbstractClass role. This class is abstract and forms the root of a hierarchy of classes in which all leaf classes play the ConcreteClass role;
2. there is at least one class playing the ConcreteClass role, and every class playing this role is a concrete subclass of the class which plays the AbstractClass role;
3. the class playing the AbstractClass role contains at least one method which plays the PrimitiveOperation role and all such methods are defined (i.e. not implemented) methods;
4. the class playing the AbstractClass role contains precisely one method which plays the TemplateMethod role. This method is implemented and contains at least one self invocation to a method which plays the PrimitiveOperation role;
5. every method which plays the PrimitiveOperation role in a class playing the ConcreteClass role is implemented.

The first of these properties is exactly analogous, up to the names of the classes involved, to the first property of the Mediator pattern (see Section 2.1). However, in this case the pattern has no classes outside this hierarchy so we do not need to explicitly exclude other classes; the third parameter of the *hierarchy* function is therefore the empty set. In addition, we do not need to specify explicitly that the AbstractClass class is abstract because this is implied by property 3 – a class which contains a defined method cannot be concrete. Our specification of property 1 is therefore written entirely in terms of the *hierarchy* function:

**value**
  AbstractClass_hierarchy : Wf_Design_Renaming → **Bool**
  AbstractClass_hierarchy(dr) ≡
    hierarchy(AbstractClass, {ConcreteClass}, {}, dr)

Similarly, the second property is exactly analogous, again up to the names of the classes involved, to the fifth property of the Mediator pattern:

**value**
  exists_concrete_class : Wf_Design_Renaming → **Bool**
  exists_concrete_class(dr) ≡ exists_role(ConcreteClass, dr),

  is_ConcreteClass : Wf_Design_Renaming → **Bool**
  is_ConcreteClass(dr) ≡
    is_concrete(AbstractClass, ConcreteClass, dr)

The third and fourth properties have no counterparts in the Mediator pattern since they are properties of methods. The third is written simply using the functions *has_def_method* and *has_all_def_method* from [4]. The first of these checks that all classes playing a given role contain a defined method playing a given role, while the second checks that all methods playing a given role in some class are defined methods.

**value**
has_PrimitiveOperation_def : Wf_Design_Renaming → **Bool**
has_PrimitiveOperation_def(dr) ≡
has_def_method(AbstractClass, PrimitiveOperation, dr) ∧
has_all_def_method(AbstractClass, PrimitiveOperation, dr)

The fourth property is specified using the functions *has_impl_method*, *unique_method* and *request_primitives*. The first of these is analogous to the function *has_def_method* used above except that it checks for an implemented method instead of a defined method. The function *unique_method* checks that all classes playing a given role contain precisely one method which plays a given role. In addition, it checks that no subclass of the given class contains more than one method playing the given role. This second part is not useful in the Template Method pattern but is used in, for example, the Observer pattern to check that the method playing the Update role is unique both in the Observer class and in the ConcreteObserver classes (see Section 2.7). The third function, *request_primitives*, checks that all methods playing a particular role in some given class are implemented and include in their bodies an invocation to self of a method playing another given role. With the appropriate parameters below, this is precisely the second part of the fourth property of the Template Method pattern.

**value**
has_TemplateMethod_impl : Wf_Design_Renaming → **Bool**
has_TemplateMethod_impl(dr) ≡
has_impl_method(AbstractClass, TemplateMethod, dr) ∧
request_primitives
(AbstractClass, TemplateMethod, PrimitiveOperation, dr) ∧
unique_method(AbstractClass, TemplateMethod, dr)

The final property is checked using the function *has_all_impl_method.* This is analogous to the function *has_all_def_method* used in the specification of property 3 above except that it checks that all methods are implemented.

**value**
all_PrimitiveOperation_impl : Wf_Design_Renaming → **Bool**
all_PrimitiveOperation_impl(dr) ≡
has_all_impl_method(ConcreteClass, PrimitiveOperation, dr)

Combining these properties together yields the following function which verifies if a design represents a Template Method pattern.

**value**
  is_template_method : Wf_Design_Renaming → **Bool**
  is_template_method(dr) ≡
    AbstractClass_hierarchy(dr) ∧
    exists_concrete_class(dr) ∧
    is_ConcreteClass(dr) ∧
    has_TemplateMethod_impl(dr) ∧
    has_PrimitiveOperation_def(dr) ∧ all_PrimitiveOperation_impl(dr)

## 2.3 State Pattern

The State pattern, which is also known as Objects for States, encapsulates the state of an object in other, separate objects [2]. In this way it encapsulates state-dependent behaviour: the ConcreteState classes represent the different state-dependent behaviours. It is used "when an object's behaviour depends on its state and it must change its behaviour at run-time depending on that state, or when operations have large, multipart conditional statements that depend on the object's state" [5]. Its structure is shown in Figure 4.

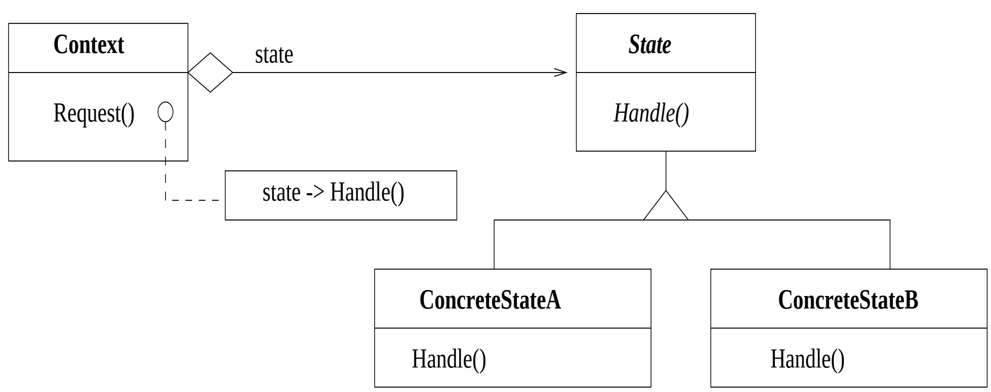


Figure 4: State Pattern Structure

The intent, participants and collaborations of the pattern are defined in [5] as follows:

### Intent

Allow an object to alter its behaviour when its internal state changes. The object will appear to change its class.

### Participants

**Context**

- defines the interface of interest to clients.
- maintains an instance of a ConcreteState subclass that defines the current state.

**State**

- defines an interface for encapsulating the behaviour associated with a particular state of the Context.

**ConcreteState**

- each subclass implements a behaviour associated with a state of the Context.

### Collaborations

- Context delegates state-specific requests to the current ConcreteState object.
- A context may pass itself as an argument to the State object handling the request. This lets the State object access the context if necessary.
- Context is the primary interface for clients. Clients can configure a context with State objects. Once a context is configured, its clients don't have to deal with the State objects directly.
- Either Context or the ConcreteState subclasses can decide which state succeeds another and under what circumstances.

The structure of the State pattern comprises a single hierarchy of classes rooted at the abstract State class, together with a single Context class. The Context class basically defines a common interface which clients can use to interact with the various ConcreteState subclasses, and essentially it simply forwards requests appropriately via its state variable. This is represented by the single aggregation relation between these classes in the pattern structure.

In practice, therefore, clients generally interact with the Context class rather than with the State classes directly, although there can and probably will be direct interaction, in particular when the state variable in the Context class is being instantiated. This instantiation of the state variable is not defined explicitly in the pattern, however, so we omit it from our analysis and specification.

In fact there is no Client class shown in the pattern structure at all, although certain responsibilities of clients are mentioned in the participants and the collaborations. We therefore do not specify that a client class must exist, though in practice a design using the State pattern would surely contain at least one such class otherwise the pattern can hardly be claimed to have its intended use. We do model one property of clients, however, namely that all client classes have either an association or aggregation relation with the Context class and interact with it via its Request interface.

Finally, just as for the Mediator pattern (see Section 2.1), the State pattern can have different variations in its structure which can tailor it to provide a more concrete or comprehensive solution to a particular problem. In fact seven different variations of the pattern can be found in [2]. Again, we do not consider such variations here but will address them in future work.

### 2.3.1 Formal Specification of the State Pattern

The State pattern uses the following names for classes, state variables and methods:

**value**
Context : G.Class_Name,
State : G.Class_Name,
ConcreteState : G.Class_Name,
Client : G.Class_Name,
state : G.Variable_Name,
Handle : G.Method_Name,
Request : G.Method_Name

The classes and relations in the State pattern satisfy the following properties:

1. there is a single class which plays the State role. This class is abstract and forms the root of a hierarchy of classes in which all leaf classes play the ConcreteState role and in which no class plays either the Context role or the Client role;
2. there is a single class which plays the Context role and this class is concrete;
3. the class which plays the Context role contains a single state variable which plays the state role;
4. the class playing the Context role is linked to the class playing the State role by a one-one aggregation relation representing the state variable playing the state role, and there are no other relations between these two classes;
5. there is at least one class playing the ConcreteState role, and every class playing this role is a concrete subclass of the class which plays the State role;
6. the class playing the State role contains at least one method which plays the Handle role and all such methods are defined methods;
7. every method which plays the Handle role in a class playing the ConcreteState role is implemented;
8. the class playing the Context role contains at least one method which plays the Request role. All methods which play this role are implemented and contain at least one invocation to the state variable playing the state role of a method which plays the Handle role;
9. any class playing the Client role is linked to the class playing the Context role by an association or aggregation relation, and the client classes communicate with the context by invoking its Request methods.

The first of these properties is analogous to the first property of the Mediator pattern (see Section 2.1), up to the names of the classes of course, but as in the Template Method pattern (see Section 2.2) we do not need to specify explicitly that the State class is abstract because according to property 6 the class contains a defined method and therefore cannot be concrete. We therefore specify only the properties of the hierarchy:

**value**
  State_hierarchy : Wf_Design_Renaming → **Bool**
  State_hierarchy(dr) ≡
    hierarchy(State, {ConcreteState}, {Context, Client}, dr)

The second property is specified using the functions *exists_one* and *is_concrete_class* from [4]. The function *exists_one* checks that a single class in the design plays a given role in the pattern, and the function *is_concrete_class* checks that all classes that play a given role are concrete.

**value**
  exists_one_Context : Wf_Design_Renaming → **Bool**
  exists_one_Context(dr) ≡ exists_one(Context, dr),

  is_concrete_Context : Wf_Design_Renaming → **Bool**
  is_concrete_Context(dr) ≡ is_concrete_class(Context, dr),

Properties 3, 4 and 5 are analogous to properties 3, 4 and 5 of the Mediator pattern. The specifications of these properties therefore follow the corresponding specifications given in Section 2.1.

**value**
  context_relationship : Wf_Design_Renaming → **Bool**
  context_relationship(dr) ≡
    has_assoc_aggr_var_ren
      (Context, State, Aggregation, state, G.one, dr) ∧
    has_unique_assoc_aggr(Context, State, dr) ∧
    store_unique_vble(Context, state, dr),

  exists_ConcreteState : Wf_Design_Renaming → **Bool**
  exists_ConcreteState(dr) ≡ exists_role(ConcreteState, dr),

  is_ConcreteState : Wf_Design_Renaming → **Bool**
  is_ConcreteState(dr) ≡ is_concrete(State, ConcreteState, dr)

The sixth property is identical up to the class and method names to the third property of the Template Method pattern (see Section 2.2).

```
value
  has_Handle_def : Wf_Design_Renaming → Bool
  has_Handle_def(dr) ≡
    has_def_method(State, Handle, dr) ∧
    has_all_def_method(State, Handle, dr)
```

Property 7 is analogous to property 5 of the Template Method pattern.

```
value
  all_Handle_impl : Wf_Design_Renaming → Bool
  all_Handle_impl(dr) ≡
    has_all_impl_method(ConcreteState, Handle, dr)
```

The eighth property is new. The fact that a class has a particular method (which may be inherited) is specified using the function *exists_method*, while the function *deleg_with_var* checks the remainder of the property.

```
value
  inside_Request : Wf_Design_Renaming → Bool
  inside_Request(ds, r) ≡
    exists_method(Context, Request, r) ∧
    deleg_with_var(Context, Request, state, State, Handle, (ds, r))
```

The final property is also new and is checked using the functions *has_assoc_aggr_reltype* and *use_interface*. These check the two clauses of the final property respectively.

```
value
  client_relationship : Wf_Design_Renaming → Bool
  client_relationship(dr) ≡
    has_assoc_aggr_reltype(Client, Context, AssAggr, G.one, dr) ∧
    use_interface(Client, Context, Request, dr)
```

These properties are then combined together to give the following function which checks if a design represents a State pattern:

```
value
  is_state_pattern : Wf_Design_Renaming → Bool
  is_state_pattern(dr) ≡
```

is_concrete_Context(dr) ∧
exists_one_Context(dr) ∧
State_hierarchy(dr) ∧
exists_ConcreteState(dr) ∧
client_relationship(dr) ∧
context_relationship(dr) ∧
is_ConcreteState(dr) ∧
has_Handle_def(dr) ∧ all_Handle_impl(dr) ∧ inside_Request(dr)

## 2.4 Strategy Pattern

The Strategy pattern, which is also known as the Policy pattern, is used "when many related classes differ only in their behaviour; when you need different variants of an algorithm, for example reflecting different space/time trade-offs; when an algorithm uses data that clients shouldn't know about; or when a class defines many behaviours and these appear as multiple conditional statements in its operations" [5]. The OMT diagram in Figure 5 shows the structure of the Strategy pattern.

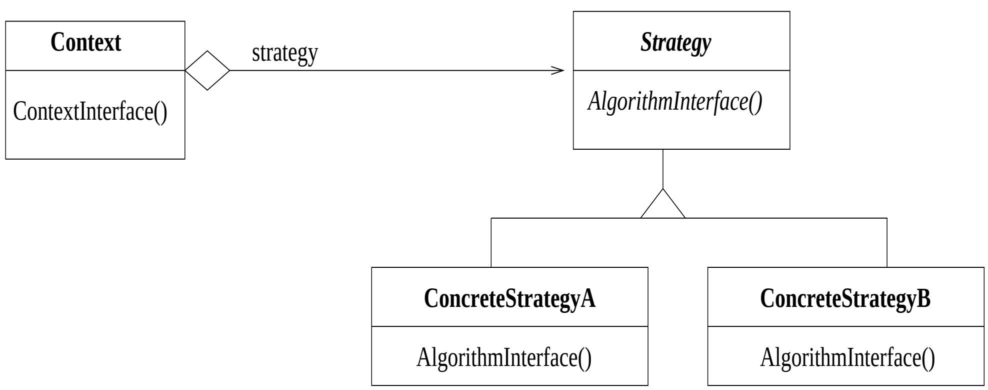


Figure 5: Strategy Pattern Structure

The intent, participants and collaborations of the pattern are defined as follows in [5]:

### Intent

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

### Participants

#### Context

- is configured with a ConcreteStrategy object.

- maintains a reference to a Strategy object.
- may define an interface that lets Strategy access its data.

**Strategy**

- declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy.

**ConcreteStrategy**

- implements the algorithm using the Strategy interface.

**Collaborations**

- Strategy and Context interact to implement the chosen algorithm. A context may pass all data required by the algorithm to the strategy when the algorithm is called. Alternatively, the context can pass itself as an argument to Strategy operations. That lets the strategy call back on the context as required.
- A context forwards requests from its clients to its strategy. Clients usually create and pass a ConcreteStrategy object to the context; thereafter, clients interact with the context exclusively. There is often a family of ConcreteStrategy classes for a client to choose from.

Although the intent of the Strategy pattern is completely different from that of the State pattern, the two patterns have almost identical structures: up to renaming of the entities appearing in the pattern, the only difference is that in the State pattern there is an annotation accompanying its Request method whereas in the Strategy pattern there is no such annotation accompanying the corresponding ContextInterface method.

In fact, even though this annotation is not present in the Strategy pattern, it is clear from the intent and collaborations of the pattern that the ContextInterface method must in any case contain an invocation to the strategy state variable of the AlgorithmInterface method. The effective properties of the two patterns, at least as defined by their structure, are therefore identical.

We can, however, identify one important difference between the two patterns which is not apparent from the structure but which derives from the intent. The intent of the Strategy pattern is that the Strategy class hierarchy provides alternative implementations of a single algorithm, with the AlgorithmInterface method in each ConcreteStrategy subclass representing a particular variation of the algorithm [11]. This means there should be only one method in the design having a renaming to AlgorithmInterface. In the State pattern, on the other hand, the intent of the State class hierarchy is to describe changes in the behaviour of an object which depend on its state. This description of the behaviour could of course involve many methods, so in the State pattern there may be more than one method in the design which renames to Handle.

### 2.4.1 Formal Specification of the Strategy Pattern

The names of the classes, methods and variables used in the Strategy pattern, and indeed the whole specification of the pattern barring the additional property identified above, can be obtained immediately by simply applying a formal renaming to the specification of the State pattern. Note that where the names used in the two patterns are the same no formal renaming is required. Note also that we additionally rename the function *is_state_pattern* to *is_strategy0* since this does not represent the whole specification of the Strategy pattern.

**use**
  Strategy **for** State,
  ConcreteStrategy **for** ConcreteState,
  AlgorithmInterface **for** Handle,
  ContextInterface **for** Request,
  strategy **for** state,
  is_strategy0 **for** is_state_pattern
**in**
  STATE

The additional property of the Strategy pattern identified above is stated as:

1. the class playing the Strategy role contains exactly one method which plays the AlgorithmInterface role

This property is embodied in the function *unique_method* and is specified as follows:

**value**
  unique_method_algorithm_interface : Wf_Design_Renaming → **Bool**
  unique_method_algorithm_interface(dr) ≡
    unique_method(Strategy, AlgorithmInterface, dr)

Then a simple conjunction of the functions *unique_method_algorithm_interface* and *is_strategy0* gives the function which checks whether a given design matches the Strategy pattern:

**value**
  is_a_strategy : Wf_Design_Renaming → **Bool**
  is_a_strategy(dr) ≡
    unique_method_algorithm_interface(dr) ∧ is_strategy0(dr)

## 2.5 Iterator Pattern

The Iterator pattern, which is also known as the Cursor pattern, is used "to access an aggregate object's contents without exposing its internal representation; to support multiple traversals of aggregate objects; or to provide a uniform interface for traversing different aggregate structures (that is, to support polymorphic iteration)" [5]. Its structure is shown in Figure 6.

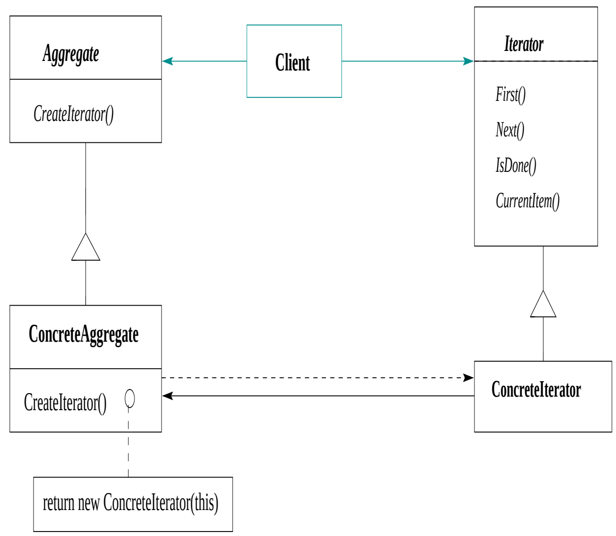


Figure 6: Structure of the Iterator Pattern

The intent, participants and collaborations of the pattern defined in [5] are:

**Intent**

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

**Participants**

**Iterator**

- defines an interface for accessing and traversing elements.

**ConcreteIterator**

- implements the Iterator interface.
- keeps track of the current position in the traversal of the aggregate.

**Aggregate**

- defines an interface for creating an Iterator object.

**ConcreteAggregate**

- implements the Iterator creation interface to return an instance of the proper ConcreteIterator.

**Collaborations**

- A ConcreteIterator keeps track of the current object in the aggregate and can compute the succeeding object in the traversal.

The structure again consists of two linked hierarchies, similar to the Mediator pattern (see Section 2.1), though the relations linking the hierarchies are different: in the Iterator pattern a ConcreteAggregate class creates instances of ConcreteIterator classes using its CreateIterator methods and the ConcreteIterator class then acts back on the same ConcreteAggregate class, this class having passed itself as parameter to the instantiation. There should thus be one CreateIterator method in a ConcreteAggregate class for each instantiation relation linking that class to a ConcreteIterator class, and each ConcreteIterator class should have association relations with precisely those ConcreteAggregate classes that instantiate it.

Note that we also require that there are no "redundant" ConcreteIterator classes, just as we required there to be no redundant ConcreteColleague classes in the Mediator pattern. Again, this is not strictly necessary but a ConcreteIterator class which is not related to any ConcreteAggregate classes cannot carry out its responsibilities in the pattern which we consider violates the spirit of the pattern.

Although the ConcreteIterator class is shown as an empty class in the structure of the pattern (see Figure 6), it clearly must implement the four methods First, Next, IsDone and CurrentItem that it inherits from the Iterator class. We therefore include these methods in our specification, which is based on the modified structure shown in Figure 7.

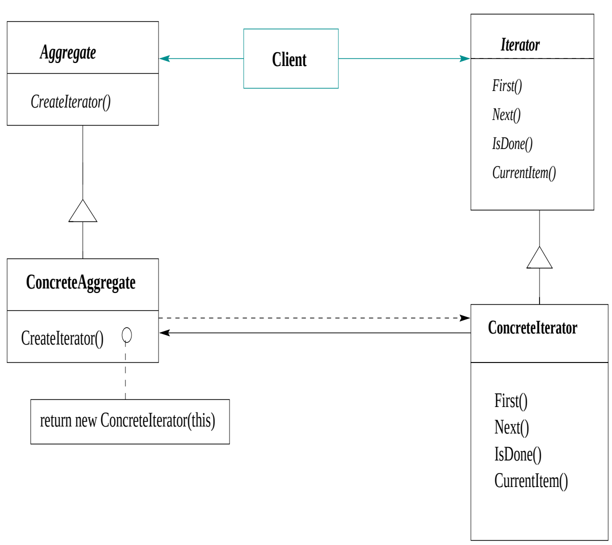


Figure 7: The Modified Iterator Structure

Finally, note also that we do not explicitly specify any properties of clients. This is because the Client class is drawn in the structure in [5] in a form which indicates that it has no specific responsibilities in the pattern.

### 2.5.1 Formal Specification of the Iterator Pattern

We begin as usual by defining the names of the entities appearing in the pattern. These are:

**value**
Iterator : G.Class_Name,
ConcreteIterator : G.Class_Name,
Aggregate : G.Class_Name,
ConcreteAggregate : G.Class_Name,
CreateIterator : G.Method_Name,
First : G.Method_Name,
Next : G.Method_Name,
IsDone : G.Method_Name,
CurrentItem : G.Method_Name

Now we describe the properties of each of the classes that participates in the pattern:

1. there is a single class which plays the Aggregate role. This class is abstract and forms the root of a hierarchy of classes in which all leaf classes play the ConcreteAggregate role and in which no class plays either the Iterator role or the ConcreteIterator role;
2. there is a single class which plays the Iterator role. This class is abstract and forms the root of a hierarchy of classes in which all leaf classes play the ConcreteIterator role and in which no class plays either the Aggregate role or the ConcreteAggregate role;
3. the class playing the Aggregate role contains at least one method which plays the CreateIterator role and all such methods are defined methods;
4. the class playing the Iterator role contains exactly one method which plays the First role and this is a defined method;
5. the class playing the Iterator role contains exactly one method which plays the Next role and this is a defined method;
6. the class playing the Iterator role contains exactly one method which plays the IsDone role and this is a defined method;
7. the class playing the Iterator role contains exactly one method which plays the CurrentItem role and this is a defined method;

8. there is at least one class playing the ConcreteAggregate role, and every class playing this role is a concrete subclass of the class which plays the Aggregate role;
9. there is at least one class playing the ConcreteIterator role, and every class playing this role is a concrete subclass of the class which plays the Iterator role;
10. the method which plays the First role is implemented in every class which plays the ConcreteIterator role;
11. the method which plays the Next role is implemented in every class which plays the ConcreteIterator role;
12. the method which plays the IsDone role is implemented in every class which plays the ConcreteIterator role;
13. the method which plays the CurrentItem role is implemented in every class which plays the ConcreteIterator role;
14. the number of methods which play the CreateIterator role in a class playing the ConcreteAggregate role is equal to the number of instantiation relations linking that class to classes playing the ConcreteIterator role. Each such method is implemented and contains an instantiation with parameter self of one of the classes playing the ConcreteIterator role with which its class is linked by an instantiation relation, and the result of this instantiation is the result of the method. Each method corresponds to a different instantiation relation so no two methods contain the same instantiation (that is an instantiation of the same class);
15. for every class playing the ConcreteIterator role there is at least one class playing the ConcreteAggregate role which is linked to it by an instantiation relation;
16. instantiation and association relations between classes playing the ConcreteAggregate and ConcreteIterator roles come in pairs. Thus, if a ConcreteAggregate class has an instantiation relation to a ConcreteIterator class then there must be an association relation between the same two classes but in the opposite direction. Similarly, if a ConcreteIterator class has an association relation with a ConcreteAggregate class then there must be an instantiation relation between the same two classes but in the opposite direction.

The first two properties are analogous to property 1 of the Template Method pattern (see Section 2.2) and again we only need to specify the properties of the class hierarchies because both the Aggregate class and the Iterator class contain defined methods (properties 3 to 7) and so must be abstract.

**value**
  Aggregate_hierarchy : Wf_Design_Renaming $\rightarrow$ **Bool**
  Aggregate_hierarchy(dr) $\equiv$
    hierarchy
      (

Aggregate, {ConcreteAggregate}, {Iterator, ConcreteIterator}, dr
),

Iterator_hierarchy : Wf_Design_Renaming → **Bool**
Iterator_hierarchy(dr) ≡
hierarchy
(
Iterator, {ConcreteIterator}, {Aggregate, ConcreteAggregate}, dr
)

The third property is analogous to property 3 of the TemplateMethod pattern (see Section 2.2):

**value**
has_CreateIterator_def : Wf_Design_Renaming → **Bool**
has_CreateIterator_def(dr) ≡
has_def_method(Aggregate, CreateIterator, dr) ∧
has_all_def_method(Aggregate, CreateIterator, dr)

Properties 4 to 7 are analogous to a combination of the first parts of properties 3 and 4 of the Template Method pattern (see Section 2.2). We therefore specify these properties using the functions *has_def_method* and *unique_method*:

**value**
has_First_def : Wf_Design_Renaming → **Bool**
has_First_def(dr) ≡
unique_method(Iterator, First, dr) ∧
has_def_method(Iterator, First, dr),

has_Next_def : Wf_Design_Renaming → **Bool**
has_Next_def(dr) ≡
unique_method(Iterator, Next, dr) ∧
has_def_method(Iterator, Next, dr),

has_IsDone_def : Wf_Design_Renaming → **Bool**
has_IsDone_def(dr) ≡
unique_method(Iterator, IsDone, dr) ∧
has_def_method(Iterator, IsDone, dr),

has_CurrentItem_def : Wf_Design_Renaming → **Bool**
has_CurrentItem_def(dr) ≡
unique_method(Iterator, CurrentItem, dr) ∧
has_def_method(Iterator, CurrentItem, dr)

The eighth and ninth properties are analogous to property 5 of the Mediator pattern (see Section 2.1:

**value**
  exists_ConcreteAggregate : Wf_Design_Renaming → **Bool**
  exists_ConcreteAggregate(dr) ≡ exists_role(ConcreteAggregate, dr),

  is_concrete_Aggregate : Wf_Design_Renaming → **Bool**
  is_concrete_Aggregate(dr) ≡
    is_concrete(Aggregate, ConcreteAggregate, dr),

  exists_ConcreteIterator : Wf_Design_Renaming → **Bool**
  exists_ConcreteIterator(dr) ≡ exists_role(ConcreteIterator, dr),

  is_concrete_Iterator : Wf_Design_Renaming → **Bool**
  is_concrete_Iterator(dr) ≡
    is_concrete(Iterator, ConcreteIterator, dr)

Properties 10 through 13 are specified simply using the function *has_impl_method* which was used in the specification of the fourth property of the Template Method pattern (see Section 2.2).

**value**
  has_First_impl : Wf_Design_Renaming → **Bool**
  has_First_impl(dr) ≡
    has_impl_method(ConcreteIterator, First, dr),

  has_Next_impl : Wf_Design_Renaming → **Bool**
  has_Next_impl(dr) ≡
    has_impl_method(ConcreteIterator, Next, dr),

  has_IsDone_impl : Wf_Design_Renaming → **Bool**
  has_IsDone_impl(dr) ≡
    has_impl_method(ConcreteIterator, IsDone, dr),

  has_CurrentItem_impl : Wf_Design_Renaming → **Bool**
  has_CurrentItem_impl(dr) ≡
    has_impl_method(ConcreteIterator, CurrentItem, dr)

The functions *nro_inst_asso*, *res_loc_vchge_inst_aparam_self* and *different_create_iterator* are used to check the three parts of property 14 respectively. The first two of these functions correspond exactly to the properties required. The function *different_create_iterator* actually uses the knowledge about the result and body of the method which is incorporated in the function *res_loc_vchge_inst_aparam_self* to simplify the specification. We know from this function that the CreateIterator function is implemented and that its result is a variable representing the requisite instantiation in its body. Therefore, we can use the results of the different methods to access their respective instantiations and it then suffices to ensure that these instantiations are different in different CreateIterator methods: since the parameter to each of these instantiations is the same, namely self, the instantiations can only be different if the classes they instantiate are different.

**value**
  has_CreateIterator_impl : Wf_Design_Renaming → **Bool**
  has_CreateIterator_impl(dr) ≡
    res_loc_vchge_inst_aparam_self
      (ConcreteAggregate, CreateIterator, ConcreteIterator, dr) ∧
    different_create_iterator
      (ConcreteAggregate, CreateIterator, dr) ∧
    nro_inst_asso
      (ConcreteAggregate, ConcreteIterator, CreateIterator, dr)

Property 15 is checked directly using the function *has_instantiation*.

**value**
  ConcreteAggregate_creates_iterator : Wf_Design_Renaming → **Bool**
  ConcreteAggregate_creates_iterator(dr) ≡
    has_instantiation(ConcreteAggregate, ConcreteIterator, dr)

The function *equal_inst_asso* is used to verify the final property. This again corresponds exactly to the property required.

**value**
  ConcreteAggregate_Iterator : Wf_Design_Renaming → **Bool**
  ConcreteAggregate_Iterator(dr) ≡
    equal_inst_asso(ConcreteAggregate, ConcreteIterator, dr)

All these properties are then combined in the standard way to give the following function which verifies whether or not a design matches the Iterator pattern:

**value**
  is_an_iterator : Wf_Design_Renaming → **Bool**
  is_an_iterator(dr) ≡

Aggregate_hierarchy(dr) ∧
Iterator_hierarchy(dr) ∧
exists_ConcreteAggregate(dr) ∧
exists_ConcreteIterator(dr) ∧
is_concrete_Aggregate(dr) ∧
is_concrete_Iterator(dr) ∧
has_First_def(dr) ∧
has_Next_def(dr) ∧
has_IsDone_def(dr) ∧
has_CurrentItem_def(dr) ∧
has_First_impl(dr) ∧
has_Next_impl(dr) ∧
has_IsDone_impl(dr) ∧
has_CurrentItem_impl(dr) ∧
has_CreateIterator_def(dr) ∧
has_CreateIterator_impl(dr) ∧
ConcreteAggregate_creates_iterator(dr) ∧
ConcreteAggregate_Iterator(dr)

## 2.6 Memento Pattern

The Memento pattern, which is also known as the Token pattern, is used when "a snapshot of (some portion of) an object's state must be saved so that the object can be restored to that state later, but a direct interface to obtaining the state would expose implementation details and break the object's encapsulation" [5]. The structure of the pattern is shown in Figure 8.

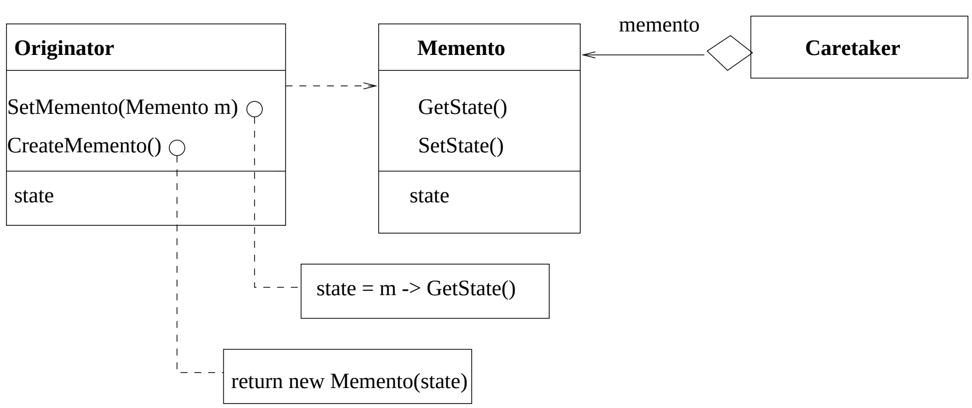


Figure 8: Memento Pattern Structure

The intent, participants and collaborations of the pattern are defined as follows in [5]:

### Intent

Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

**Participants**

**Memento**

- stores internal state of the Originator object. The memento may store as much or as little of the originator's internal state as necessary at its originator's discretion.
- protects against access by objects other than the originator. Mementos have effectively two interfaces. Caretaker sees a narrow interface to the Memento – it can only pass the memento to other objects. Originator, in contrast, sees a wide interface, one that lets it access all the data necessary to restore itself to its previous state. Ideally, only the originator that produced the memento would be permitted to access the memento's internal state.

**Originator**

- creates a memento containing a snapshot of its current internal state.
- uses the memento to restore its internal state.

**Caretaker**

- is responsible for the memento's safekeeping.
- never operates on or examines the contents of a memento.

**Collaborations**

- A caretaker requests a memento from an originator, holds it for a time, and passes it back to the originator, as illustrated in the interaction diagram in Figure 9. Sometimes the caretaker will not pass the memento back to the originator, because the originator might never need to revert to an earlier state.
- Mementos are passive. Only the originator that created a memento will assign or retrieve its state.

In the pattern structure shown in Figure 8 the Memento class contains GetState and SetState methods which are both shown "undefined", that is without annotations or parameters. The reason for this may be because there are in practice different ways of implementing these methods. In the case of SetState, at the one extreme we could have a single method which instantiates all state variables in the Memento class at once, and at the other extreme we could have one SetState method for each state variable, with anything in between these two extremes being also a possibility (i.e. several SetState methods which instantiate some but not all of the state variables). In the case of GetState, a similar range of alternative implementations can be envisaged, and again we could have a single GetState method which returns all the state variables in the Memento class at once, or one GetState method for each state variable, or something intermediate between these two extremes.

While it would be possible to specify these various alternatives, we feel that having only one SetState method and one GetState method is most in-keeping with the spirit of the pattern:

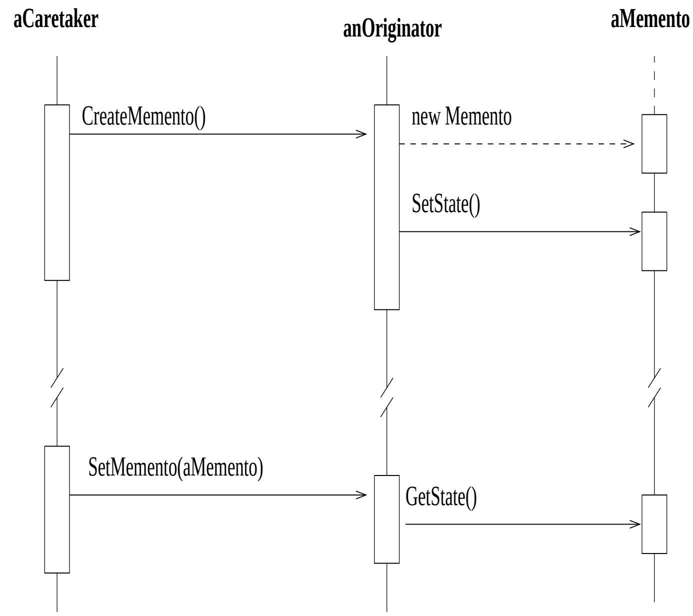


Figure 9: Collaborations of the Memento Pattern

the Originator effectively sees and treats the Memento as a single entity and does not need to know about or manipulate its internal structure. We therefore choose to model a single SetState method which instantiates all state variables in the Memento class at the same time and a single GetState method which returns all the state variables in the Memento class at once, say in the form of some sort of tuplet. With this choice, the number of parameters of the SetState method should be the same as the number of state variables in the Memento class, and the GetState method requires no parameters (in fact it requires no parameters in all the alternative implementations). In addition, we can define the body of the SetMemento method to consist of a simple assignment to all state variables in the Originator class of an invocation of the single GetState method on the memento. We make these properties explicit by adding appropriate annotations to the methods as shown in the modified structure in Figure 10.

There are also (at least) two alternative implementations of the body of the CreateMemento method: the Memento class could implement a parameterised new method which creates a new instance of the class and sets its state variables at the same time, or this process could be done in two steps using first the basic unparameterised new method to create the instance and then instantiating the state variables using the SetState method. In this case we choose the latter alternative, primarily because otherwise the SetState method effectively performs no useful function in the pattern. The appropriate annotation is modified accordingly in the modified pattern structure shown in Figure 10.

The Caretaker has an important role in the Memento pattern as described in the collaborations above: it causes the Originator to create a memento by invoking its CreateMemento method, and it can also cause it to reset its state to that stored in the memento by invoking its SetMemento method. In the first of these interactions, the Originator invokes the CreateMemento method to

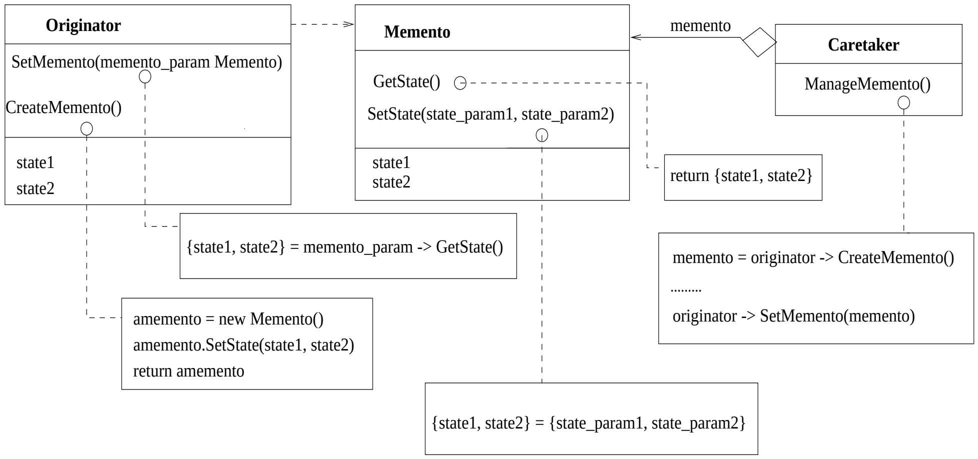


Figure 10: The Modified Memento Structure

create a new memento and set the state of that memento by "copying" the appropriate parts of its own state to it using the SetState method (see the annotation to the CreateMemento method in Figure 10). In the second interaction, the Originator retrieves the state of its memento using the SetMemento method.

In an actual implementation, it is possible that these two interactions take place singly within different methods in the Caretaker or in sequence in a single method, probably with other interactions in between them. We make these two possibilities explicit in the pattern by introducing three new method roles in the Caretaker class: RenewMemento, which represents a method which contains the first interaction (i.e. which causes the Originator to invoke its CreateMemento method); ResetMemento, which represents a method which contains the second interaction (i.e. which causes the Originator to invoke its SetMemento method); and ManageMemento, which represents a method which contains both interactions in the appropriate order. For simplicity we show only the method ManageMemento in the modified structure diagram in Figure 10 though we note that this could be replaced by the two methods RenewMemento and ResetMemento and indeed in our specification below we allow either of these two alternatives.

Finally, note that we do not explicitly model a relationship between the Originator and the Caretaker classes even though such a relation is perhaps implied by the interaction diagram in Figure 9 because in some cases there may be no such relationship. For example the Originator could be a singleton class (i.e. a class with a single instance as represented by the Singleton Pattern in [5]), in which case the Caretaker would simply reference the sole instance of the class directly.

### 2.6.1 Formal Specification of the Memento Pattern

The following constants are relevant to the Memento pattern:

**value**
Originator : G.Class_Name,
Memento : G.Class_Name,
Caretaker : G.Class_Name,
memento : G.Variable_Name,
memento_param : G.Variable_Name,
state : G.Variable_Name,
originator : G.Variable_Name,
state_param : G.Variable_Name,
SetMemento : G.Method_Name,
CreateMemento : G.Method_Name,
GetState : G.Method_Name,
SetState : G.Method_Name,
RenewMemento : G.Method_Name,
ResetMemento : G.Method_Name,
ManageMemento : G.Method_Name

The classes and relations in the Memento pattern satisfy the following properties:

1. there is a single class which plays the Memento role and this class is concrete;
2. there is a single class which plays the Originator role and this class is concrete;
3. there is a single class which plays the Caretaker role and this class is concrete;
4. the class which plays the Memento role contains one or more state variables which play the state role;
5. the class which plays the Originator role contains one or more state variables which play the state role;
6. the class which plays the Caretaker contains a single state variable which plays the memento role;
7. the class playing the Caretaker role is linked to the class playing the Memento role by a one-one aggregation relation representing the state variable playing the memento role, and there are no other relations between these two classes;
8. the class playing the Originator role and the class playing the Memento role have the same number of state variables which play the state role;
9. the class playing the Memento role contains exactly one method which plays the GetState role. This method is implemented, has no parameters, and its result is the set of all state variables which play the state role;

10. the class playing the Memento role contains precisely one method which plays the SetState role. This method is implemented and returns no result, and the number of its parameters is equal to the number of state variables which play the state role in the Memento class. Each parameter plays the state_param role and the body of the method assigns each state variable which plays the state role to a different parameter;

11. the class playing the Originator role contains a unique method which plays the CreateMemento role and the body of this method consists of an instantiation with no parameters of the class playing the Memento role followed by an invocation of the method which plays the SetState role on the instance created. The parameters of this invocation are all the state variables which play the state role, and the result of this invocation is the result of the CreateMemento method;

12. the class playing the Originator role contains a unique method which plays the SetMemento role. This method is implemented and has no result and only one parameter, this parameter playing the memento_param role and its type being the class which plays the Memento role. The body of the method consists of a single invocation to the parameter of the method which plays the GetState role in the class which plays the Memento role, and the results of this invocation are assigned to different variables playing the state role in the class playing the Originator role;

13. the class playing the Originator role is linked to the class playing the Memento role by an instantiation relation;

14. the class playing the Caretaker role contains *either* at least one method which plays the RenewMemento role *or* at least one method which plays the ManageMemento role. Each RenewMemento method is implemented and its body includes an invocation to some variable, say $v$, of the method which plays the CreateMemento role in the Originator class and records the result of this invocation in some variable, say $vd$. For each RenewMemento method there is at least one ResetMemento method which is implemented and whose body includes an invocation to the same variable $v$ of the method which plays the SetMemento role in the Originator class, the variable $vd$ being passed as the parameter to this invocation. Each ManageMemento method is also implemented and has a body which performs both the invocations described above, in that order;

15. the class playing the Caretaker role never operates on its memento state variable.

Properties 1, 2 and 3 are analogous to property 2 of the State pattern (see Section 2.3).

**value**
exists_one_concrete_Originator : Wf_Design_Renaming → **Bool**
exists_one_concrete_Originator(pr) ≡
  exists_one(Originator, pr) ∧ is_concrete_class(Originator, pr),

exists_one_concrete_Memento : Wf_Design_Renaming → **Bool**

exists_one_concrete_Memento(pr) ≡
  exists_one(Memento, pr) ∧ is_concrete_class(Memento, pr),

exists_one_concrete_Caretaker : Wf_Design_Renaming → **Bool**
exists_one_concrete_Caretaker(pr) ≡
  exists_one(Caretaker, pr) ∧ is_concrete_class(Caretaker, pr)

Properties 4 and 5 are specified using the function *store_vble*.

**value**
  store_state : Wf_Design_Renaming → **Bool**
  store_state(pr) ≡
    store_vble(Originator, state, pr) ∧
    store_vble(Memento, state, pr)

Properties 6 and 7 are analogous to properties 3 and 4 of the State pattern.

**value**
  Caretaker_relation : Wf_Design_Renaming → **Bool**
  Caretaker_relation(pr) ≡
    has_assoc_aggr_var_ren
      (Caretaker, Memento, Aggregation, memento, G.one, pr) ∧
    has_unique_assoc_aggr(Caretaker, Memento, pr) ∧
    store_unique_vble(Caretaker, memento, pr)

Property 8 is equivalent to saying that the Memento class is the memory of the Originator class or alternatively that it has a copy of the Originator state variables. The property is new and is specified using the function *quantities_of_variables*, which simply states that any two classes playing the given two roles contain the same number of state variables playing a third given role.

**value**
  copy_of_state : Wf_Design_Renaming → **Bool**
  copy_of_state(ds, r) ≡
    quantities_of_variables(Originator, Memento, state, r)

The majority of property 9 is also new. The functions *unique_method* and *has_impl_method* are used to check that the GetState method is implemented and unique, as in property 4 of the Template Method pattern (see Section 2.2). The fact that the method has no parameters is checked using the function *no_parameter_in_design*, and the function *result_of_Get_State* checks that the result of the method is the set of all state variables which play the state role.

**value**
  M_has_GetState_impl : Wf_Design_Renaming → **Bool**
  M_has_GetState_impl(pr) ≡
    unique_method(Memento, GetState, pr) ∧
    has_impl_method(Memento, GetState, pr) ∧
    no_parameter_in_design(Memento, GetState, pr) ∧
    result_of_Get_State(Memento, GetState, state, pr)

The tenth property is also largely new. The uniqueness of the SetState method is checked using the function *unique_method*, while the function *has_method_without_res* checks that all methods which play a given role have an empty result. The fact that all parameters of the SetState method play the same state_param role is checked using the function *all_pars_same_ren*, which more generally checks that all parameters in a given method play a single given role, while the function *params_vars_in_SetState_one* checks the remainder of the property.

**value**
  M_has_SetState_impl : Wf_Design_Renaming → **Bool**
  M_has_SetState_impl(pr) ≡
    unique_method(Memento, SetState, pr) ∧
    has_method_without_res(Memento, SetState, pr) ∧
    params_vars_in_SetState_one(Memento, SetState, state, pr) ∧
    all_pars_same_ren
      (Memento, SetState, state_param, pr)

The function *unique_method* is also used to check the uniqueness of the CreateMemento method in property 11. The function *res_local_var_change_inst_aparam_ren* checks the remainder of this property.

**value**
  O_has_createM_interface : Wf_Design_Renaming → **Bool**
  O_has_createM_interface(pr) ≡
    res_local_var_change_inst_aparam_ren
      (Originator, CreateMemento, state, Memento, SetMemento, pr) ∧
    unique_method(Originator, CreateMemento, pr)

The twelfth property is specified using the function *unique_method* again, together with the functions *one_image_ren_pars_in_design*, *SetM_state_var_change_deleg_par* and *has_method_without_res*.

The first of these checks that a method has only one parameter and that the type of the parameter is some given class. The second checks that the SetMemento method is implemented and its body assigns its state variables to the results of applying the GetState method to its parameter. And the third checks that the result of the SetMemento method is empty.

```
value
  O_has_setM_interface : Wf_Design_Renaming → Bool
  O_has_setM_interface(pr) ≡
    one_image_ren_pars_in_design
      (Originator, SetMemento, Memento, memento_param, pr) ∧
    SetM_state_var_change_deleg_par
      (
        Originator,
        SetMemento,
        state,
        memento_param,
        Memento,
        GetState,
        pr
      ) ∧
    has_method_without_res(Originator, SetMemento, pr) ∧
    unique_method(Originator, SetMemento, pr)
```

Because of the uniqueness of the classes playing the Originator and the Memento roles, property 13 is in fact equivalent to property 11 of the Iterator pattern.

```
value
  ori_mem_relation : Wf_Design_Renaming → Bool
  ori_mem_relation(pr) ≡ has_instantiation(Originator, Memento, pr)
```

The existence of the methods in property 14 and the fact that they are all implemented is checked by the functions *has_impl_method* and *has_all_impl_method* in the normal way. The functions *assign_invoke_param1* and *assign_invoke_param2* check the properties of the bodies of ManageMemento and of RenewMemento and its corresponding ResetMemento respectively, and the function *never_operate* checks property 15. This last function in fact checks the more general property that one given class has no method in which there is an invocation to a variable representing either an association or aggregation relation with another class.

```
value
  Caretaker : Wf_Design_Renaming → Bool
  Caretaker(pr) ≡
```

(has_impl_method(Caretaker, RenewMemento, pr) ∨
 has_impl_method(Caretaker, ManageMemento, pr)) ∧
has_all_impl_method(Caretaker, RenewMemento, pr) ∧
has_all_impl_method(Caretaker, ResetMemento, pr) ∧
has_all_impl_method(Caretaker, ManageMemento, pr) ∧
assign_invoke_param1(
    Caretaker, memento, Originator, ManageMemento,
    CreateMemento, SetMemento, pr) ∧
assign_invoke_param2(
    Caretaker, memento, Originator, RenewMemento,
    ResetMemento, CreateMemento, SetMemento, pr) ∧
never_operate(Caretaker, Memento, pr)

Combining all these properties together yields the following specification of a function which checks whether a design represents a Memento pattern:

**value**
  is_memento_pattern : Wf_Design_Renaming → **Bool**
  is_memento_pattern(dr) ≡
    exists_one_concrete_Caretaker(dr) ∧
    exists_one_concrete_Originator(dr) ∧
    exists_one_concrete_Memento(dr) ∧
    Caretaker_relation(dr) ∧
    ori_mem_relation(dr) ∧
    store_state(dr) ∧
    copy_of_state(dr) ∧
    M_has_SetState_impl(dr) ∧
    M_has_GetState_impl(dr) ∧
    Caretaker(dr) ∧
    O_has_createM_interface(dr) ∧ O_has_setM_interface(dr)

## 2.7 Observer Pattern

The Observer pattern, which is also known both as Dependents and as Publish-Subscribe, is used when an abstraction has two aspects, one dependent on the other; when a change to one object requires changing others and you don't know how many objects need to be changed; and when an object should be able to notify other objects without making assumptions about what those objects are" [5]. It is basically an abstraction of the well-known dependency mechanism used in Smalltalk's Model-View-Controller architecture in which all views of the model are notified when the state of the model is modified [8].

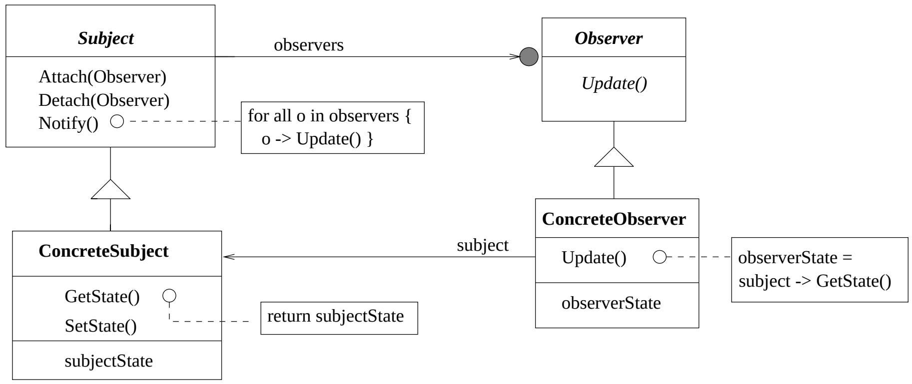


Figure 11: Observer Pattern Structure

The OMT diagram in Figure 11 shows the Observer structure.

The intent, participants and collaborations of the pattern are defined in [5] as follows:

## Intent

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

## Participants

### Subject

- knows its observers. Any number of Observer objects may observe a subject.
- provides an interface for attaching and detaching Observer objects.

### Observer

- defines an updating interface for objects that should be notified of changes in a subject.

### ConcreteSubject

- stores state of interest to ConcreteObserver objects.
- sends a notification to its observers when its state changes.

### ConcreteObserver

- maintains a reference to a ConcreteSubject object.
- stores state that should stay consistent with the subject's.
- implements the Observer updating interface to keep its state consistent with the subject's.

## Collaborations

- ConcreteSubject notifies its observers whenever a change occurs that could make its observers' states inconsistent with its own.
- After being informed of a change in the concrete subject, a ConcreteObserver object may query the subject for information. ConcreteObserver uses this information to reconcile its state with that of the subject.

The interaction diagram in Figure 12 illustrates the collaborations between a subject and two observers. Note how the Observer object that initiates the change request postpones its update until it receives the Update message from the subject.

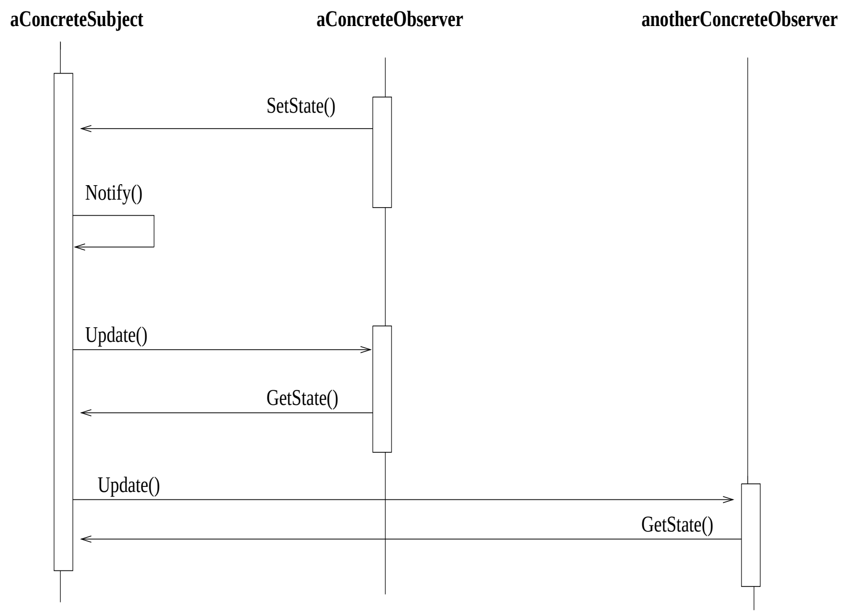


Figure 12: Observer Collaborations

The structure of the Observer pattern also consists essentially of two linked hierarchies like the Mediator and Iterator patterns (see Sections 2.1 and 2.5 respectively), but again the relationships between the hierarchies are different. Here, the Subject class records a collection of observers in its state variable observers, this being represented by the one-many aggregation relation between these two classes. Conversely, each instance of a ConcreteObserver class holds a reference in its subject state variable to a single instance of a ConcreteSubject class.

This latter relation is shown between the ConcreteObserver and ConcreteSubject classes in the structure, which suggests that each ConcreteObserver class is specific to a particular ConcreteSubject class, that is has a relationship with only one ConcreteSubject class. This is what we model, together with additional constraints which say first that each ConcreteObserver class cannot have more observerState variables than the number of subjectState variables in the ConcreteSubject class it is related to and second that there are no "redundant" ConcreteSubject classes, that is that each ConcreteSubject class is related to at least one ConcreteObserver class.

However, this is certainly not the only possibility and it is in fact debatable whether it really constitutes an abstraction of Smalltalk's Model-View-Controller mechanism. Indeed it rather seems to constitute a restriction of the MVC mechanism because in MVC it is quite acceptable for a view to be used with a range of different models. But if we want to model a situation in which ConcreteObservers can be associated with different ConcreteSubject classes it seems more appropriate to place the relationship between the Observer class and the Subject class, as is in fact the case in MVC, rather than between the corresponding concrete classes. We therefore consider this case to be one of several possible variants of the Observer pattern and will deal with it in more detail in future work.

Either way, the two relations linking the subject and observer hierarchies should in fact be reciprocal, so that the subject state variable of an observer should be precisely the subject to which it was added by the Attach method. One way in which this constraint could be maintained would be to ensure that the subject state variable of an observer is set by the Attach method. However, another possibility would of course be to instantiate the subject variable in another method which subsequently invokes the Attach method. We therefore give no specification of this consistency property below but note that without such a consistency constraint a design is not necessarily consistent with the Observer pattern.

One thing we can deduce about the Attach and Detach methods is that they must respectively add and remove an observer from the observers state variable. In our model we assume that various basic operations on collections of objects are "built-in", including operations for adding and removing an object to or from a collection of objects, which we denote respectively by the methods collectionadd and collectionremove. The bodies of the Attach and Detach methods must therefore include an invocation to the observers state variable of the appropriate one of these methods, the parameters of these invocations including the observer instance which appears as the parameter of the Attach or Detach method itself.

When a method which does not represent one of these built-in operations (for example the Update method) is invoked on a collection of objects, we interpret this in our model as meaning that the message is in fact sent to each of the objects in the collection independently. In this way, the annotation of the Notify method is modelled as a single invocation to the observers state variable of the Update method.

The SetState and GetState methods perform similar tasks to the methods with the same names in the Memento pattern (see Section 2.6). However, in the Observer pattern the ConcreteObserver objects do not necessarily store all of the subjectState variables in the ConcreteSubject's state. Moreover, different ConcreteObservers may store different parts of the ConcreteSubject's state, that is different subsets of its subjectState variables. In the Observer pattern, therefore, the ConcreteSubject's state is not treated as a single entity, as is the case of the Memento's state in the Memento pattern. It would thus not be unreasonable to have more than one SetState and GetState method in the Observer pattern, each setting or returning some subset of the subjectState variables, provided of course that each subjectState variable is set and returned by at least one of each of those methods.

We therefore assume simply that there is at least one GetState method and that each such method has no parameters and has a non-empty result which consists of some subset of the subjectState variables. Similarly, we assume that there is at least one SetState method and that each such method has no result and assigns each of its parameters, which should not be empty, to a different subjectState variable. We further assume that each subjectState variable appears in the result of at least one GetState method and is assigned in the body of at least one SetState method. These assumptions are again included as annotations in a modified OMT diagram of the pattern structure which is shown in Figure 13.

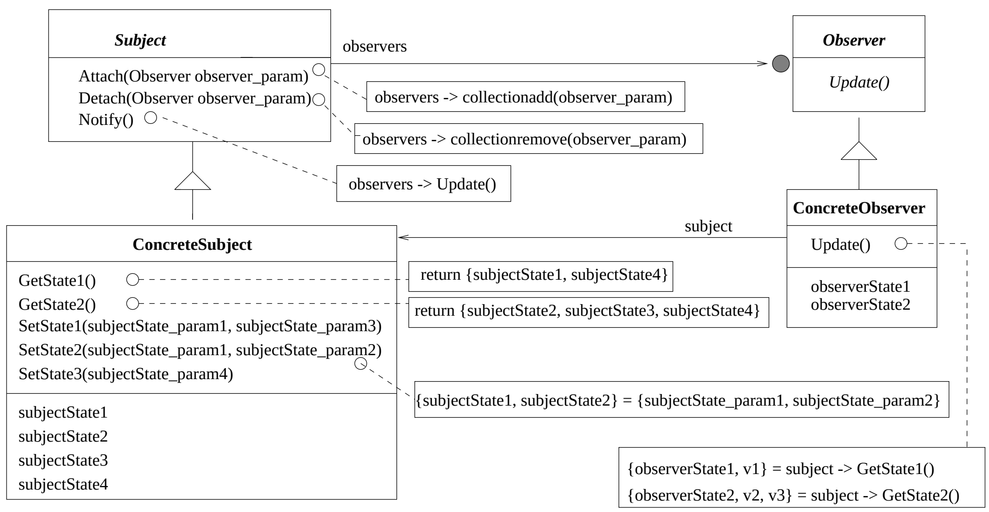


Figure 13: The Modified Observer Structure

With these assumptions, the body of the Update method is similar to the body of the SetMemento method in the Memento pattern. The main differences are that the Update method can contain more than one invocation, each to a different GetState method, and the receiver of these invocations (i.e. the subject variable) is a state variable, whereas in the SetMemento method there is only a single invocation to the GetState method and the receiver of this invocation is the parameter of the method. Note also that, as shown in Figure 13, additional local variables may be needed in the Update method because the GetState methods do not necessarily return the exact subset of the subject state that is required by each concrete observer: additional state may be returned which the observer simply ignores.

### 2.7.1 Formal Specification of the Observer Pattern

As usual, we begin by introducing constants which represent the names of the classes, methods, state variables and parameters used in the pattern.

**value**
  Subject : G.Class_Name,

ConcreteSubject : G.Class_Name,
Observer : G.Class_Name,
ConcreteObserver : G.Class_Name,
observers : G.Variable_Name,
observer_param : G.Variable_Name,
subject : G.Variable_Name,
subjectState : G.Variable_Name,
observerState : G.Variable_Name,
subjectState_param : G.Variable_Name,
Attach : G.Method_Name,
Detach : G.Method_Name,
Notify : G.Method_Name,
GetState : G.Method_Name,
SetState : G.Method_Name,
Update : G.Method_Name

The classes and relations in the Observer pattern satisfy the following properties:

1. there is a single class which plays the Subject role. This class is abstract and forms the root of a hierarchy of classes in which all leaf classes play the ConcreteSubject role and in which no class plays either the Observer role or the ConcreteObserver role;
2. there is a single class which plays the Observer role. This class is abstract and forms the root of a hierarchy of classes in which all leaf classes play the ConcreteObserver role and in which no class plays either the Subject role or the ConcreteSubject role;
3. the class playing the Subject role contains a single state variable which plays the observers role;
4. the class playing the Subject role is linked to the class playing the Observer role by a one-many association relation representing the state variable playing the observers role;
5. the class playing the Observer role contains exactly one method which plays the Update role and this method is a defined method;
6. there is at least one class playing the ConcreteSubject role, and every class playing this role is a concrete subclass of the class which plays the Subject role;
7. there is at least one class playing the ConcreteObserver role, and every class playing this role is a concrete subclass of the class which plays the Observer role;
8. the class playing the Subject role contains exactly one method which plays the Attach role. This method is implemented and has a single parameter, which plays the observer_param role and which is of type Observer. The body of the method contains an invocation to the state variable playing the observers role of the primitive collectionadd method, and the parameter of the Attach method is included in the parameters of this invocation;

9. the class playing the Subject role contains exactly one method which plays the Detach role. This method is implemented and has a single parameter, which plays the observer_param role and which is of type Observer. The body of the method contains an invocation to the state variable playing the observers role of the primitive collectionremove method, and the parameter of the Detach method is included in the parameters of this invocation;

10. the class playing the Subject role contains exactly one method which plays the Notify role. This method is implemented and its body contains an invocation to the variable playing the observers role of the Update method in the Observer class;

11. every class playing the ConcreteSubject role contains at least one state variable playing the subjectState role;

12. every class playing the ConcreteSubject role contains at least one method which plays the GetState role. Each such method is implemented, has no parameters, and returns a non-empty subset of the state variables which play the subjectState role. Each subjectState variable belongs to the result of at least one GetState method;

13. every class playing the ConcreteSubject role contains at least one method which plays the SetState role. Each such method is implemented, has no result, and has at least one parameter, and all the parameters play the subjectState_param role. The body of each method simply assigns each of the parameters to a different subjectState variable, and each subjectState variable appears in such an assignment in at least one SetState method;

14. every class playing the ConcreteObserver role has exactly one state variable which plays the subject role;

15. every class playing the ConcreteObserver role is linked to exactly one class playing the ConcreteSubject role by a one-one association relation representing the state variable playing the subject role, and there are no other relations between classes of these two roles. In addition, every class playing the ConcreteSubject role has a link of this form from at least one class playing the ConcreteObserver role;

16. every class playing the ConcreteObserver role has at least one state variable which plays the observerState role, and the number of such variables is not greater than the number of state variables playing the subjectState role in the ConcreteSubject class represented by its subject state variable;

17. the method playing the Update role is implemented in every class playing the ConcreteObserver role and returns no result. The body of the method consists of a series of assignments to subsets of the observerState state variables of the result of invoking some GetState method on the subject state variable, and each observerState state variable is set by at least one such assignment.

The first two properties are analogous to property 1 of the Mediator pattern (see Section 2.1), though as in the TemplateMethod pattern (see Section 2.2) we do not need to explicitly specify that the Observer class is abstract because this is implied by property 5.

**value**

Subject_hierarchy : Wf_Design_Renaming → **Bool**
Subject_hierarchy(dr) ≡
  hierarchy
    (Subject, {ConcreteSubject}, {Observer, ConcreteObserver}, dr),

is_abstract_Subject : Wf_Design_Renaming → **Bool**
is_abstract_Subject(dr) ≡ is_abstract_class(Subject, dr),

Observer_hierarchy : Wf_Design_Renaming → **Bool**
Observer_hierarchy(dr) ≡
  hierarchy
    (Observer, {ConcreteObserver}, {Subject, ConcreteSubject}, dr)

Properties 3 and 14 are analogous to property 3 of the Mediator pattern.

**value**

store_observers : Wf_Design_Renaming → **Bool**
store_observers(dr) ≡ store_unique_vble(Subject, observers, dr),

store_subject : Wf_Design_Renaming → **Bool**
store_subject(dr) ≡
  store_unique_vble(ConcreteObserver, subject, dr)

Aside from the cardinality of the relation, property 4 is analogous to the first part of property 4 of the Mediator pattern and is therefore specified simply using the function *has_assoc_aggr_var_ren*.

**value**

Observer_relation : Wf_Design_Renaming → **Bool**
Observer_relation(dr) ≡
  has_assoc_aggr_var_ren
    (Subject, Observer, Association, observers, G.many, dr)

Property 5 is analogous to property 4 of the Iterator pattern (see Section 2.5).

**value**

Update_defined : Wf_Design_Renaming → **Bool**
Update_defined(dr) ≡
  has_def_method(Observer, Update, dr) ∧
  unique_method(Observer, Update, dr)

Properties 6 and 7 are analogous to property 5 of the Mediator pattern.

**value**
exists_concrete_Subject : Wf_Design_Renaming → **Bool**
exists_concrete_Subject(dr) ≡ exists_role(ConcreteSubject, dr),

is_concrete_Subject : Wf_Design_Renaming → **Bool**
is_concrete_Subject(dr) ≡
is_concrete(Subject, ConcreteSubject, dr),

exists_concrete_Observer : Wf_Design_Renaming → **Bool**
exists_concrete_Observer(dr) ≡ exists_role(ConcreteObserver, dr),

is_concrete_Observer : Wf_Design_Renaming → **Bool**
is_concrete_Observer(dr) ≡
is_concrete(Observer, ConcreteObserver, dr)

The first two parts of property 8 are analogous to the first two parts of property 4 of the TemplateMethod pattern. The properties of the parameter are analogous to the properties of the parameter in the SetMemento method (see property 12 of the Memento pattern in Section 2.6) so are similarly specified using the function *one_image_ren_pars_in_design*. The properties of the body of the method are new and are specified by the function *deleg_with_var_coll_aparam_ren*. This checks more generally that the body of every method playing a given role in some class contains an invocation to each state variable playing a given role of a given (built-in) collection method, the parameters of the invocation being those playing some given role. In this particular case we know from other properties that there is only one parameter to the Attach method and only ony observers state variable, so this specifies precisely the required body of the Attach method.

**value**
Attach_implemented : Wf_Design_Renaming → **Bool**
Attach_implemented(dr) ≡
unique_method(Subject, Attach, dr) ∧
has_impl_method(Subject, Attach, dr) ∧
one_image_ren_pars_in_design
(Subject, Attach, Observer, observer_param, dr) ∧
deleg_with_var_coll_aparam_ren
(
Subject, Attach, observers, G.collectionadd, observer_param, dr
)

Property 9 is entirely analogous to property 8 above except that the collectionremove method not the collectionadd method is invoked in the Detach method.

**value**

Detach_implemented : Wf_Design_Renaming → **Bool**
Detach_implemented(dr) ≡
  unique_method(Subject, Detach, dr) ∧
  has_impl_method(Subject, Detach, dr) ∧
  one_image_ren_pars_in_design
    (Subject, Detach, Observer, observer_param, dr) ∧
  deleg_with_var_coll_aparam_ren
    (
      Subject,
      Detach,
      observers,
      G.collectionremove,
      observer_param,
      dr
    )

The first two parts of property 10 are again analogous to the first two parts of property 4 of the TemplateMethod pattern, while the properties of the body of the Notify method are analogous to the properties of the body of the Request method in the State pattern (see Section 2.3) and so are similarly specified using the function *deleg_with_var*.

**value**

Notify_implemented : Wf_Design_Renaming → **Bool**
Notify_implemented(dr) ≡
  has_impl_method(Subject, Notify, dr) ∧
  deleg_with_var
    (Subject, Notify, observers, Observer, Update, dr) ∧
  unique_method(Subject, Notify, dr)

Property 11 is analogous to property 4 of the Memento pattern (see Section 2.6). However, we do not need to specify this property explicitly here because it is implied by property 12: this states that there is at least one GetState method and each such method returns a non-empty subset of the subjectState state variables; therefore there must be at least one subjectState state variable.

Property 12 is similar to property 9 of the Memento pattern (see Section 2.6).The main difference here is that in the Observer pattern we can have more than one GetState method and that each one returns a non-empty subset of the subjectState variables. We therefore use the function *has_all_impl_method* instead of *unique_method*, and we use the new function *results_of_Get_State* to define the properties of the results of the methods. In addition to checking that the result of each GetState method is a non-empty subset of the subjectState state variables, this function also checks that each subjectState state variable appears in the result of at least one GetState method.

**value**
GetState_implemented : Wf_Design_Renaming → **Bool**
GetState_implemented(dr) ≡
has_impl_method(ConcreteSubject, GetState, dr) ∧
has_all_impl_method(ConcreteSubject, GetState, dr) ∧
no_parameter_in_design(ConcreteSubject, GetState, dr) ∧
results_of_Get_State
(ConcreteSubject, GetState, subjectState, dr)

Property 13 is similarly largely analogous to property 10 of the Memento pattern. Here, however, we use *has_impl_method* instead of *unique_method* because there can be more than one SetState method, and we use *params_vars_in_SetState_many* instead of *params_vars_in_SetState_one* because each SetState method may deal with only a subset of the state.

**value**
SetState_implemented : Wf_Design_Renaming → **Bool**
SetState_implemented(dr) ≡
has_impl_method(ConcreteSubject, SetState, dr) ∧
params_vars_in_SetState_many(
ConcreteSubject, SetState, subjectState, dr) ∧
all_pars_same_ren(
ConcreteSubject, SetState, subjectState_param, dr) ∧
has_method_without_res(ConcreteSubject, SetState, dr)

The first part of property 15 is specified exactly using the function *has_assoc_var_ren* and the uniqueness of the relation is specified using *has_unique_assoc_aggr* as in, for example, property 4 of the Mediator pattern. The last part of the property is checked directly by the function *class_connected*.

**value**
concrete_Observer_relation : Wf_Design_Renaming → **Bool**
concrete_Observer_relation(dr) ≡
has_assoc_var_ren
(ConcreteObserver, ConcreteSubject, subject, G.one, dr) ∧
has_unique_assoc_aggr(ConcreteObserver, ConcreteSubject, dr) ∧
class_connected(ConcreteSubject, ConcreteObserver, dr)

The first part of property 16 is analogous to property 11, though in this case we must specify it. We of course use the function *store_vble* as in property 4 of the Memento pattern. The remainder of this property is checked precisely using the function *less_quantities_of_variables*.

**value**
store_ObserverState : Wf_Design_Renaming → **Bool**
store_ObserverState(dr) ≡
store_vble(ConcreteObserver, observerState, dr) ∧
less_quantities_of_variables
(
ConcreteObserver,
observerState,
ConcreteSubject,
subjectState,
dr
)

Property 17 is new and is checked using the function *Update_state_var_change_deleg_var*.

**value**
Update_implemented : Wf_Design_Renaming → **Bool**
Update_implemented(dr) ≡
Update_state_var_change_deleg_var
(
ConcreteObserver,
Update,
observerState,
subject,
ConcreteSubject,
GetState,
dr
)

Finally, we combine all the above properties together to obtain the following specification of a function which checks whether a design represents an Observer pattern:

**value**
is_observer_pattern : Wf_Design_Renaming → **Bool**
is_observer_pattern(dr) ≡
Subject_hierarchy(dr) ∧
Observer_hierarchy(dr) ∧

exists_concrete_Subject(dr) $\wedge$
exists_concrete_Observer(dr) $\wedge$
Observer_relation(dr) $\wedge$
concrete_Observer_relation(dr) $\wedge$
store_subject(dr) $\wedge$
store_observers(dr) $\wedge$
store_ObserverState(dr) $\wedge$
is_abstract_Subject(dr) $\wedge$
Attach_implemented(dr) $\wedge$
Detach_implemented(dr) $\wedge$
Notify_implemented(dr) $\wedge$
GetState_implemented(dr) $\wedge$
SetState_implemented(dr) $\wedge$
Update_defined(dr) $\wedge$
Update_implemented(dr) $\wedge$
is_concrete_Observer(dr) $\wedge$ is_concrete_Subject(dr)

## 2.8 Command Pattern

The Command pattern is used to parameterize objects such as menu items with an action they are supposed to perform; to specify, queue, and execute requests at different times; to support undo operations; to support the logging of changes so that they can be reapplied; or to structure a system around high-level operations built from primitive operations. It is also known as both Action and Transaction. Its structure is shown in Figure 14.

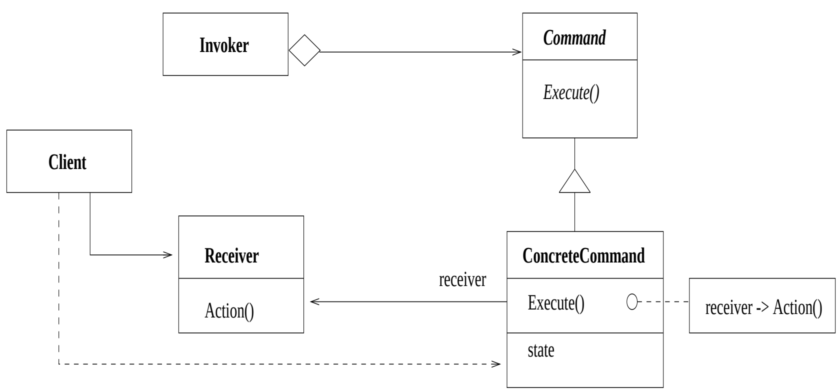


Figure 14: Command Pattern Structure

Command objects define operations on components and other objects. They are similar to messages in traditional object-oriented systems. They can also be executed in isolation to perform arbitrary computation, and they can reverse the effects of such execution to support undo. [13].

Some commands may be directly accessible to the user while others are only used internally in a system. For example, operations offered on a menu are directly accessible to the user, but after choosing a particular item from a menu, say to print a document, the system may enter a mode where the user is offered further choices, for example which document to print or which printer to print it on. In addition, either the command or the operands could be chosen first. Thus, for example, in a text editor a region of text may first be selected, then a command such as copy or delete may be given [7].

The intent, participants and collaborations of the pattern are defined in [5] as follows:

**Intent**

Encapsulate a request as an object, thereby letting you parameterise clients with different requests, queue or log requests, and support undoable operations.

**Participants**

**Command**

- declares an interface for executing an operation.

**ConcreteCommand**

- defines a binding between a Receiver object and an action.
- implements Execute by invoking the corresponding operation(s) on Receiver.

**Client**

- creates a ConcreteCommand object and sets its receiver.

**Invoker**

- asks the command to carry out the request.

**Receiver**

- knows how to perform the operations associated with carrying out a request. Any class may serve as a Receiver.

**Collaborations**

- The client creates a ConcreteCommand object and specifies its receiver.
- An Invoker object stores the ConcreteCommand object.
- The invoker issues a request by calling Execute on the command. When commands are undoable, ConcreteCommand stores state for undoing the command prior to invoking Execute.
- The ConcreteCommand object invokes operations on its receiver to carry out the request.

These collaborations are illustrated in the interaction diagram in Figure 15, which also shows how the Command pattern decouples the invoker from the receiver and hence from the request it carries out.

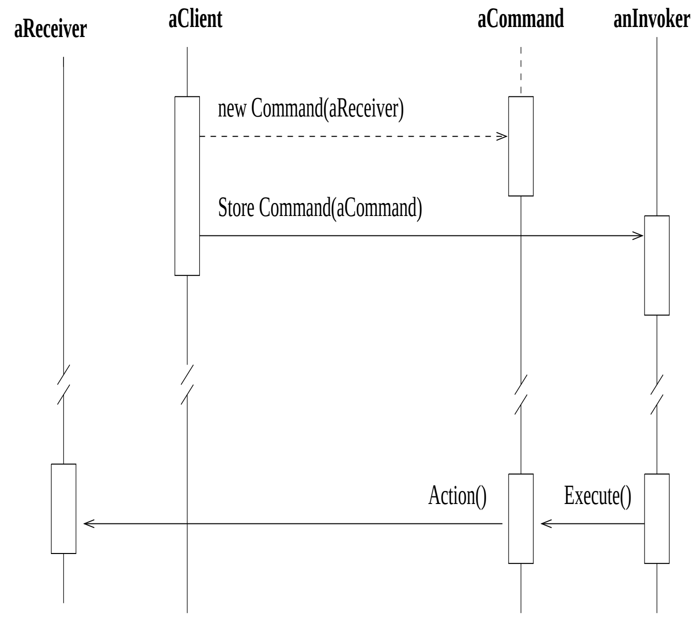


Figure 15: Collaborations of the Command Pattern

The structure of the Command pattern consists of a hierarchy of Command and ConcreteCommand classes which represent the commands, together with Invoker, Receiver and Client classes. Each ConcreteCommand class has an Execute method, which defines what the command does, and is associated with a particular Receiver class, which represents the objects on which the command acts. The command stores an instance of its Receiver class in its receiver state variable, and the command is executed by invoking a particular Action method from the Receiver class on this state variable as shown in the annotation to the Execute method.

There can be several different classes playing the Invoker role. For example, if a Command pattern is used to implement some part of a user interface toolkit (see the discussion of the motivation of the Command pattern in [5] and Figure 16) it can include different widgets that issue requests, for instance menu items and buttons.

The application in Figure 16 also shows that there can also be several different classes playing the Receiver role, the receiver basically depending on the particular command that is issued. For example, the receiver of a PasteCommand is a Document while the receiver of an OpenCommand is an Application.

Note also that in this example the Application class plays both the Client role and the Receiver role. In this situation, the association relationship linking the Client and Receiver classes shown in Figure 14 would in fact not be present in the design.

The client has an important role in the Command pattern as shown in the interaction diagram in Figure 15 and basically acts as coordinator between the Invoker, Receiver and ConcreteCommand classes. First, it creates a new concrete command and instantiates its receiver, then it passes

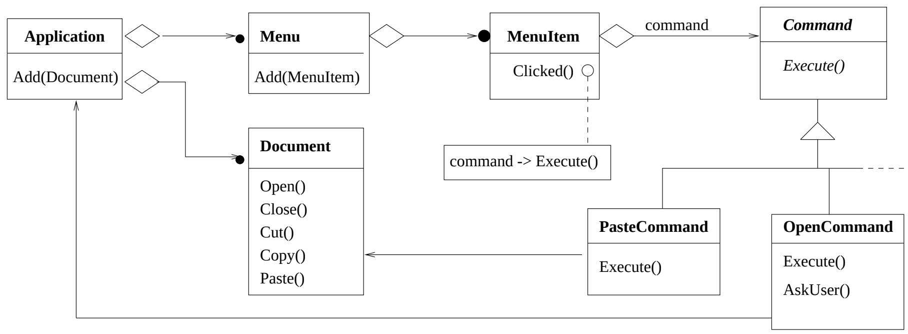


Figure 16: An Application of the Command Pattern

this concrete command to the invoker. However, the second of these interactions, that is the interaction between the client and the invoker, is not included in the OMT diagram representing the pattern structure (Figure 14).

Part of this interaction is shown in the OMT diagram representing the application of the Command pattern (Figure 16), though the Client (Application) is related to the Invoker (MenuItem) indirectly through the intermediate Menu class rather than directly and the transfer of the command from the client to the invoker is still omitted.

In our treatment of the Command pattern we include the full invocation as shown in the interaction diagram in Figure 15, and require that the Client class contains a method which invokes the StoreCommand method in the Invoker, though we allow this invocation to be indirect. We introduce a new role, ClientMethod, into the pattern to represent the method in the Client class which initiates this invocation and extend the structure of the pattern to include both this method and the StoreCommand method, as well as a (possibly) transitive relationship between the Client and the Invoker. The StoreCommand method has a single command as its parameter and simply assigns this parameter to its command state variable. The modified OMT diagram is shown in Figure 17.

Note that in this diagram we have omitted the state variable state from the ConcreteCommand class, and indeed we also omit it from our analysis and specification. According to the discussion of the Command pattern in [5], this variable is intended to support undo and redo operations. However, if such operations are to be supported additional methods are required in the pattern. For example, if commands are undoable the Command class requires an unexecute method, and if it supports command logs it will also require load and store methods.

In fact, none of these methods are included in the pattern structure in Figure 14, and indeed the Command pattern can be used to design systems which do not have these operations and which therefore do not require the state variable in the ConcreteCommand class. We therefore prefer to omit this variable, thereby restricting our specification to a Command pattern without undo

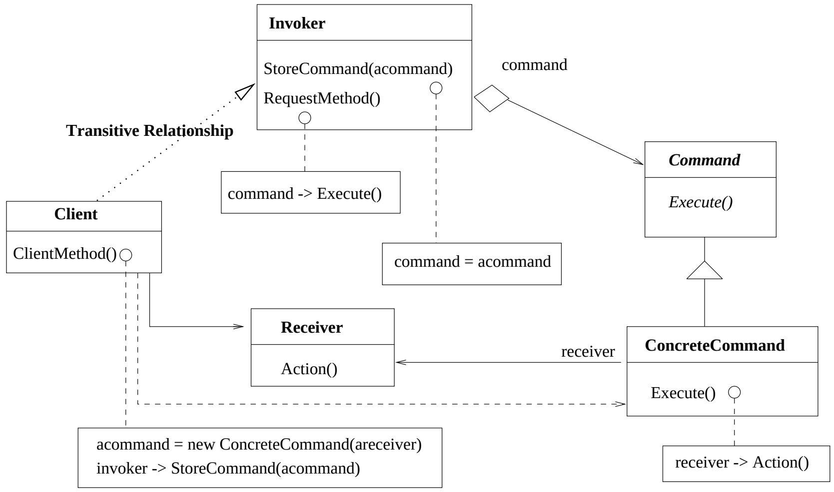


Figure 17: The Modified Command Structure

operations, and to consider the Command pattern with undo operations as a variant of this basic Command pattern. The Command pattern with undo operations will then be considered along with other pattern variants in future work.

We do consider one variation of the pattern here, though, largely because it can be incorporated as a simple, and effectively optional extension of the basic pattern. This is the *macro command*, which is introduced in [5] as a sort of "generalised" command.

Basically, a macro command represents a sequence of commands (which may itself include macro commands) and execution of such a command is equivalent to executing each command in the sequence in turn. A macro command therefore has no explicit receiver in the same sense as a normal command, but instead stores the sequence of commands which it represents. In the case of a macro command, therefore, the ClientMethod has a slightly different form than that shown in Figure 17: the instantiation of the MacroCommand class which replaces the instantiation of the ConcreteCommand class has no parameter.

We represent this macro command by extending the pattern structure with a new role MacroCommand and also extending the Command hierarchy in the pattern to allow both ConcreteCommand and MacroCommand classes as leaves. The MacroCommand class then has an aggregation relation with the abstract Command class which represents its sequence of commands. The extension to the pattern structure is shown in Figure 18.

Introducing the MacroCommand class into the Command hierarchy makes the hierarchy more complicated than the simple form, introduced and described in Section 2.1, which has featured in all the patterns we have dealt with so far. For the leaves of the hierarchy, there is really not much difference between the two forms of hierarchy – each leaf class in the Command hierarchy

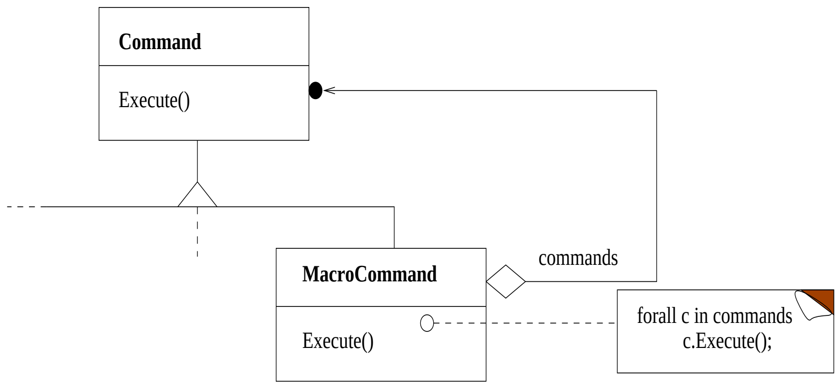


Figure 18: The Structure of the Macro Command

can simply play either, but not both, of the possible roles, ConcreteCommand or MacroCommand. However, classes intermediate between the Command class and the leaves of the hierarchy may also play either the ConcreteCommand or the MacroCommand role (again not both), and these two classes have different properties. We therefore need to ensure that neither one of these classes inherits from (is a subclass of) the other, otherwise the subclass effectively has the properties of both roles.

On the basis of the above considerations, we now proceed to give a formal specification of the properties of the Command pattern in RSL.

### 2.8.1 Formal Specification of the Command Pattern

The names of the classes, methods and variables appearing in the Command pattern are:

**value**

Invoker : G.Class_Name,
Client : G.Class_Name,
Command : G.Class_Name,
ConcreteCommand : G.Class_Name,
Receiver : G.Class_Name,
MacroCommand : G.Class_Name,
command : G.Variable_Name,
receiver : G.Variable_Name,
commands : G.Variable_Name,
command_param : G.Variable_Name,
Execute : G.Method_Name,
Action : G.Method_Name,
StoreCommand : G.Method_Name,
ClientMethod : G.Method_Name,

RequestMethod : G.Method_Name

The classes and relations in the Command pattern satisfy the following properties:

1. there is a single class which plays the Command role. This class is abstract and forms the root of a hierarchy of classes in which all leaf classes play either the ConcreteCommand role or the MacroCommand role and in which no class plays either the Invoker role, the Receiver role or the Client role. Intermediate classes may also play the ConcreteCommand role or the MacroCommand role, subject to the constraint that no class playing either one of these roles may be a subclass of a class playing the other role;
2. the class playing the Command role contains exactly one method which plays the Execute role and this method is a defined method;
3. there is at least one class playing the ConcreteCommand role, and every class playing this role is a concrete subclass of the class which plays the Command role;
4. there is at least one class playing the Invoker role and all classes playing this role are concrete;
5. every class playing the Invoker role has exactly one state variable which plays the command role;
6. every class playing the Invoker role is linked to the class playing the Command role by an aggregation relation representing the state variable playing the command role, and there are no other relations between classes of these two roles;
7. every class playing the Invoker role contains at least one method playing the Request-Method role. Each such method is implemented and its body contains an invocation to the command state variable of the method which plays the Execute role in the Command class;
8. there is at least one class playing the Receiver role and all classes playing this role are concrete;
9. there is exactly one class playing the Client role and this class is concrete;
10. the class playing the Client role has an association or aggregation relation with every class playing the Receiver role, except that if the Client class also plays the Receiver role it does not need to have such a relation with itself;
11. every class playing the ConcreteCommand role contains exactly one state variable playing the receiver role;
12. every class playing the Receiver role contains at least one method playing the Action role and each such method is implemented;

13. every class playing the ConcreteCommand role contains at least one method playing the Execute role. Each such method is implemented and its body contains an invocation to the receiver state variable of the method which plays the Action role in the Receiver class;

14. every class playing the ConcreteCommand role is linked to exactly one class playing the Receiver role by a one-one association relation representing the state variable playing the receiver role, and every class playing the Receiver role has a link of this form from at least one class playing the ConcreteCommand role;

15. every class playing the Invoker role contains exactly one method playing the StoreCommand role. This method is implemented, has no result, and has a single prameter which is of type Command and which plays the command_param role. The body of the method simply assigns this parameter to the command state variable;

16. the class playing the Client role contains at least one method playing the ClientMethod role. Each such method is implemented and contains in its body an instantiation of a class playing either the ConcreteCommand or the MacroCommand role. The instantiation of the MacroCommand role receives no parameters, while the instantiation of the ConcreteCommand role receives a single parameter which is generally a variable representing a relation between the Client class and some class playing the Receiver role but which can also be self if the Client class also plays the Receiver role. The result of this instantiation is assigned to a variable, and this variable is then passed as the sole parameter to an invocation of the StoreCommand method in some class playing the Invoker role, this invocation being possibly indirect;

17. every class playing the MacroCommand role is a concrete subclass of the class which plays the Command role;

18. every class playing the MacroCommand role contains a single state variable which plays the commands role;

19. the method playing the Execute role is implemented in every class playing the MacroCommand role, and the body of this method contains an invocation to the commands state variable of the method which plays the Execute role in the Command class;

20. every class playing the MacroCommand role is linked to the class playing the Command role by a one-many aggregation relation representing the state variable playing the commands role, and there are no other association or aggregation relations between classes of these two roles.

The first part of property 1 is analogous to property 1 of the Mediator pattern (see Section 2.1), and again as in the TemplateMethod pattern (see Section 2.2) we do not need to explicitly specify that the Command class is abstract because this is implied by property 2. The second part of the property is in fact also included in the function *hierarchy* which was introduced in Section 2.1, though in that and other patterns this additional part is automatically true because there is only one available role for the subclasses and leaf classes. The specification of property 1 therefore looks entirely analogous to the specification of the properties of the hierarchies of the other patterns.

**value**

Command_hierarchy : Wf_Design_Renaming → **Bool**
Command_hierarchy(dr) ≡
  hierarchy
    (
      Command,
      {ConcreteCommand, MacroCommand},
      {Receiver, Invoker, Client},
      dr
    )

Property 2 is analogous to property 4 of the Iterator pattern (see Section 2.5).

**value**

has_Execute_def : Wf_Design_Renaming → **Bool**
has_Execute_def(dr) ≡
  has_def_method(Command, Execute, dr) ∧
  unique_method(Command, Execute, dr)

Property 3 is analogous to property 5 of the Mediator pattern.

**value**

exists_concrete_Command : Wf_Design_Renaming → **Bool**
exists_concrete_Command(dr) ≡ exists_role(ConcreteCommand, dr),

is_concrete_Command : Wf_Design_Renaming → **Bool**
is_concrete_Command(dr) ≡
  is_concrete(Command, ConcreteCommand, dr)

The first part of property 4 is similarly analogous to the first part of property 5 of the Mediator pattern, and the second part is analogous to the second part of property 2 of the State pattern (see Section 2.3).

**value**

exists_Invoker : Wf_Design_Renaming → **Bool**
exists_Invoker(dr) ≡
  exists_role(Invoker, dr) ∧ is_concrete_class(Invoker, dr)

Properties 5 and 6 are analogous to properties 3 and 4 of the Mediator pattern.

**value**
Invoker_has_Command : Wf_Design_Renaming $\rightarrow$ **Bool**
Invoker_has_Command(dr) $\equiv$
has_unique_assoc_aggr(Invoker, Command, dr) $\wedge$
has_assoc_aggr_var_ren
(Invoker, Command, Aggregation, command, G.one, dr) $\wedge$
store_unique_vble(Invoker, command, dr)

The seventh property is analogous to property 8 of the State pattern.

**value**
Invoker_invoke : Wf_Design_Renaming $\rightarrow$ **Bool**
Invoker_invoke(ds, r) $\equiv$
exists_method(Invoker, RequestMethod, r) $\wedge$
deleg_with_var
(Invoker, RequestMethod, command, Command, Execute, (ds, r))

Property 8 is exactly analogous to property 4.

**value**
exists_Receiver : Wf_Design_Renaming $\rightarrow$ **Bool**
exists_Receiver(dr) $\equiv$
exists_role(Receiver, dr) $\wedge$ is_concrete_class(Receiver, dr)

Property 9 is exactly analogous to property 2 of the State pattern (see Section 2.3).

**value**
exists_one_Client : Wf_Design_Renaming $\rightarrow$ **Bool**
exists_one_Client(dr) $\equiv$
exists_one(Client, dr) $\wedge$ is_concrete_class(Client, dr)

Property 10 is defined using the function *has_assoc_aggr_com*.

**value**
Client_association : Wf_Design_Renaming $\rightarrow$ **Bool**
Client_association(dr) $\equiv$ has_assoc_aggr_com(Client, Receiver, dr)

Property 11 is analogous to property 3 of the Mediator pattern.

**value**

store_receiver : Wf_Design_Renaming → **Bool**
store_receiver(dr) ≡
store_unique_vble(ConcreteCommand, receiver, dr)

Property 12 is simply specified using the function *has_impl_method*. This strictly only checks that at least one Action method is implemented, and in principle allows the class to also contain defined Action methods. However, this latter is ruled out because we know from property 8 that all classes playing the Receiver role are concrete, which means that they cannot include defined methods. All Action methods must therefore be implemented methods.

**value**

Action_in_Receiver : Wf_Design_Renaming → **Bool**
Action_in_Receiver(dr) ≡ has_impl_method(Receiver, Action, dr)

Property 13 is entirely analogous to property 7. However, in this case the Execute method is known to exist because it is inherited from the Command class, so we do not need to specify this explicitly here. The clause including *exists_method* can therefore be omitted from our specification in this case.

**value**

has_Execute_cc_interface : Wf_Design_Renaming → **Bool**
has_Execute_cc_interface(dr) ≡
deleg_with_var
(ConcreteCommand, Execute, receiver, Receiver, Action, dr)

Property 14 is similar to property 15 of the Observer pattern (see Section 2.7) except that here the specified relation is not necessarily the only relation between the two classes. We therefore do not include the function *has_unique_assoc_aggr* in the specification below.

**value**

binding_Receiver_Command : Wf_Design_Renaming → **Bool**
binding_Receiver_Command(dr) ≡
has_assoc_var_ren
(ConcreteCommand, Receiver, receiver, G.one, dr) ∧
class_connected(Receiver, ConcreteCommand, dr)

The first two parts of property 15 are analogous to the first parts of property 12 of the Memento pattern (see Section 2.6). The remaining properties, namely those of the body of the method, are specified using the function *st_com_body*.

**value**

has_StoreCommand_impl : Wf_Design_Renaming → **Bool**
has_StoreCommand_impl(dr) ≡
  has_impl_method(Invoker, StoreCommand, dr) ∧
  unique_method(Invoker, StoreCommand, dr) ∧
  one_image_ren_pars_in_design
    (Invoker, StoreCommand, Command, command_param, dr) ∧
  has_method_without_res(Invoker, StoreCommand, dr) ∧
  st_com_body(Invoker, StoreCommand, command_param, command, dr)

Property 16 is specified using the functions *exists_method*, which ensures the existence of the method as in property 8 of the State pattern, and *client_comment*, which checks the remainder of the property.

**value**

comment_of_Client : Wf_Design_Renaming → **Bool**
comment_of_Client(ds, r) ≡
  exists_method(Client, ClientMethod, r) ∧
  client_comment
    (
      Client,
      ConcreteCommand,
      Receiver,
      Invoker,
      MacroCommand,
      ClientMethod,
      StoreCommand,
      (ds, r)
    )

Property 17 is exactly analogous to the second part of property 3. The first part of property 3 does not apply to MacroCommand because the Command pattern need not include MacroCommand classes but must include ConcreteCommand classes.

**value**

is_concrete_MacroCommand : Wf_Design_Renaming → **Bool**
is_concrete_MacroCommand(dr) ≡
  is_concrete(Command, MacroCommand, dr)

Property 18 is analogous to property 3 of the Mediator pattern, and property 20 is similar to property 4 of the Mediator pattern except that here we only specify that there are no other aggregation or association relations between the classes, which means that there could also be an instantiation relation. Instead of the function *has_unique_assoc_aggr* we specify this property using *has_unique_assoc_aggr_relation* which simply says that there is only one association or aggregation relation between the two given classes.

**value**
MacroCommand_relationship : Wf_Design_Renaming → **Bool**
MacroCommand_relationship(dr) ≡
has_unique_assoc_aggr_relation(MacroCommand, Command, dr) ∧
has_assoc_aggr_var_ren
(MacroCommand, Command, Aggregation, commands, G.many, dr) ∧
store_unique_vble(MacroCommand, commands, dr)

Property 19 is entirely analogous to properties 7 and 13.

**value**
has_Execute_in_mc : Wf_Design_Renaming → **Bool**
has_Execute_in_mc(dr) ≡
deleg_with_var
(MacroCommand, Execute, commands, Command, Execute, dr)

Combining the above properties together we obtain the following function which checks whether a design represents a Command pattern:

**value**
is_command_pattern : Wf_Design_Renaming → **Bool**
is_command_pattern(dr) ≡
exists_Invoker(dr) ∧
Command_hierarchy(dr) ∧
exists_one_Client(dr) ∧
exists_concrete_Command(dr) ∧
exists_Receiver(dr) ∧
store_receiver(dr) ∧
Client_association(dr) ∧
Invoker_has_Command(dr) ∧
is_concrete_Command(dr) ∧
has_Execute_def(dr) ∧
has_Execute_cc_interface(dr) ∧
comment_of_Client(dr) ∧
Invoker_invoke(dr) ∧
Action_in_Receiver(dr) ∧

binding_Receiver_Command(dr) $\wedge$
has_StoreCommand_impl(dr) $\wedge$
is_concrete_MacroCommand(dr) $\wedge$
has_Execute_in_mc(dr) $\wedge$ MacroCommand_relationship(dr)

## 2.9 Visitor Pattern

The Visitor pattern is used "when an object structure contains many classes of objects with differing interfaces and you want to perform operations on these objects that depend on their concrete classes; when many distinct and unrelated operations need to be performed on objects in an object structure and you want to avoid "polluting" their classes with these operations; and when the classes defining an object structure rarely change but you often want to define new operations over the structure" [5]. The structure of the pattern is shown in Figure 19.

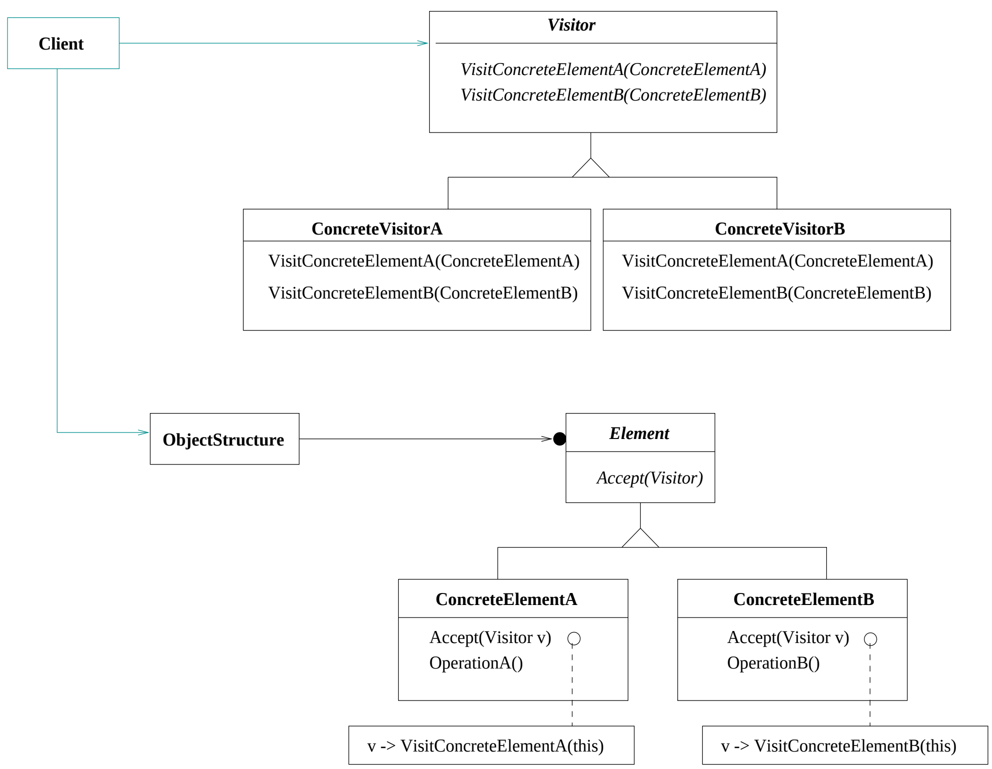


Figure 19: Visitor Pattern Structure

The intent, participants and collaborations of the pattern are defined in [5] as:

### Intent

Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

**Participants**

**Visitor**

- declares a Visit operation for each class of ConcreteElement in the object structure. The operation's name and signature identifies the class that sends the Visit request to the visitor. That lets the visitor determine the concrete class of the element being visited. Then the visitor can access the element directly through its particular interface.

**ConcreteVisitor**

- implements each operation declared by Visitor. Each operation implements a fragment of the algorithm defined for the corresponding class of object in the structure. ConcreteVisitor provides the context for the algorithm and stores its local state. This state often accumulates results during the traversal of the structure.

**Element**

- defines an Accept operation that takes a visitor as an argument.

**ConcreteElement**

- implements an Accept operation that takes a visitor as an argument.

**ObjectStructure**

- can enumerate its elements.
- may provide a high-level interface to allow the visitor to visit its elements.
- may either be a composite or a collection such as a list or a set.

**Collaborations**

- A client that uses the Visitor pattern must create a ConcreteVisitor object and then traverse the object structure, visiting each element with the visitor.
- When an element is visited, it calls the Visitor operation that corresponds to its class. The element supplies itself as an argument to this operation to let the visitor access its state, if necessary.

The interaction diagram in Figure 20 illustrates the collaborations between an object structure, a visitor, and two elements.

The basic idea of the Visitor pattern is that a composite object, which is represented by the ObjectStructure class, is constructed out of objects from the various ConcreteElement classes, and each ConcreteVisitor class performs operations on this composite object by calling appropriate sub-operations on each of the components of the object, these sub-operations being represented

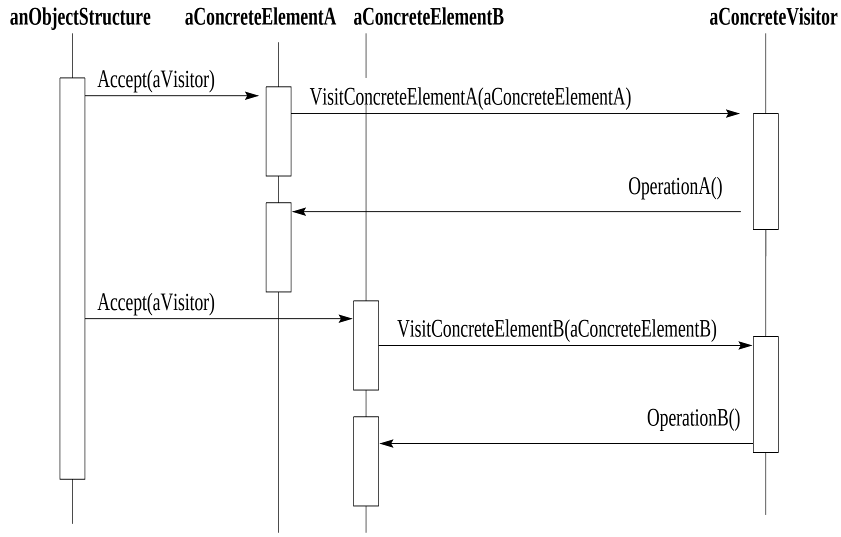


Figure 20: Collaborations of the Visitor Pattern

in the structure as the various VisitConcreteElement methods in a particular ConcreteVisitor class. Some additional post-processing of the results returned by these sub-operations may of course also be necessary. Each ConcreteVisitor class therefore contains one VisitConcreteElement method for each ConcreteElement class, and that ConcreteElement class represents the class of the object which is passed as parameter to the VisitConcreteElement method.

We restrict to a single Visitor hierarchy and a single Element hierarchy, considering without loss of generality that multiple visitors and multiple object structures in a design correspond to multiple instances of the pattern. We also assume that the body of the Accept method in the ConcreteElement classes contains only the invocation shown in the annotation in the structure. Again, it would be perfectly possible in a design for the Accept operation to do more, but we consider that situation as departing from the spirit of the pattern – if additional functionality is required it can be included in another method which first invokes the Accept operation and then implements the additional behaviour.

From the interaction diagram in Figure 20, we can see that the ObjectStructure class invokes the Accept operation on the Element class; indeed this is effectively represented by the association relation between these two classes in the structure of the pattern (Figure 19). However, the specific method which performs this invocation does not explicitly appear in the structure of the pattern. In our formal specification, however, we wish to specify the properties of this method. We therefore introduce the role SendVisitor to represent it, and we include an annotation to this method which states that its body contains an invocation of the Accept method. The OMT diagram representing the structure of the Visitor pattern is thus revised to the form shown in Figure 21 and this structure forms the basis for our analysis and specification of the pattern.

As in the Iterator pattern (see Section 2.5) we do not explicitly specify any properties of clients in the pattern because the Client class is again depicted in the structure in [5] in a form which indicates that it has no specific responsibilities.

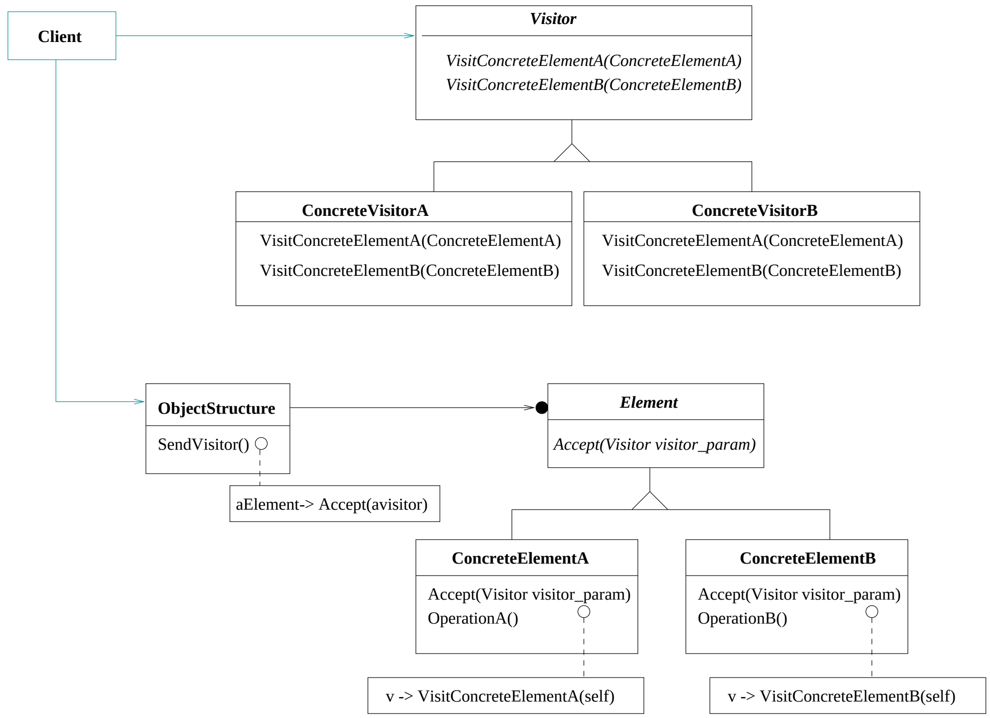


Figure 21: The Modified Visitor Structure

### 2.9.1 Formal Specification of the Visitor Pattern

The names of the entities involved in the Visitor pattern are defined by the following RSL constants:

**value**
  Visitor : G.Class_Name,
  ConcreteVisitor : G.Class_Name,
  Element : G.Class_Name,
  ConcreteElement : G.Class_Name,
  ObjectStructure : G.Class_Name,
  concreteElement : G.Variable_Name,
  visitor_param : G.Variable_Name,
  VisitConcreteElement : G.Method_Name,
  Accept : G.Method_Name,
  Operation : G.Method_Name,
  SendVisitor : G.Method_Name

Based on these considerations, we summarise the properties of the entities in the Visitor pattern as follows:

1. there is a single class which plays the Visitor role. This class is abstract and forms the root of a hierarchy of classes in which all leaf classes play the ConcreteVisitor role and in which no class plays either the Element role, the ConcreteElement role, the ObjectStructure role or the Client role;
2. there is a single class which plays the Element role. This class is abstract and forms the root of a hierarchy of classes in which all leaf classes play the ConcreteElement role and in which no class plays either the Visitor role, the ConcreteVisitor role, the ObjectStructure role or the Client role;
3. the class playing the Element role contains exactly one method which plays the Accept role. This is a defined method and has only one parameter, this parameter playing the visitor_param role and being of type Visitor;
4. there is at least one class playing the ConcreteElement role, and every class playing this role is a concrete subclass of the class which plays the Element role;
5. the class playing the Visitor role contains at least one method which plays the VisitConcreteElement role and all such methods are defined methods. Each method has a single parameter, which plays the concreteElement role and which is of type ConcreteElement, and no two methods have parameters which are of the same type;
6. the number of methods which play the VisitConcreteElement role in the class playing the Visitor role is equal to the number of classes playing the ConcreteElement role;
7. there is at least one class playing the ConcreteVisitor role, and every class playing this role is a concrete subclass of the class which plays the Visitor role;
8. every method which plays the VisitConcreteElement role in a class playing the ConcreteVisitor role is implemented;
9. every class playing the ConcreteElement role contains at least one method which plays the Operation role and all such methods are implemented methods;
10. the method playing the Accept role in each class playing the ConcreteElement role contains a unique invocation involving a method playing the VisitConcreteElement role. This invocation is sent to the parameter of the Accept method and the only parameter of the invocation is self;
11. there is a single class which plays the ObjectStructure role and this class is concrete;
12. there is either an association or an aggregation relation of cardinality one-many between the class playing the ObjectStructure role and the class playing the Element role;

13. the class playing the ObjectStructure role contains at least one method that plays the SendVisitor role and each such method is implemented and contains an invocation of the method which plays the Accept role.

Properties 1 and 2 are analogous to property 1 of the Mediator pattern (see Section 2.1), though again as in the TemplateMethod pattern (see Section 2.2) we do not need to explicitly specify that the Visitor and Element classes are abstract because this is implied by properties 3 and 5.

**value**
  Visitor_hierarchy : Wf_Design_Renaming → **Bool**
  Visitor_hierarchy(dr) ≡
    hierarchy
      (Visitor, {ConcreteVisitor}, {Element, ConcreteElement}, dr),

  Element_hierarchy : Wf_Design_Renaming → **Bool**
  Element_hierarchy(dr) ≡
    hierarchy
      (Element, {ConcreteElement}, {Visitor, ConcreteVisitor}, dr)

The third property is similar to property 8 of the Observer pattern (see Section 2.7) except that the Accept method is a defined method rather than an implemented one and furthermore its body is unspecified. We therefore omit the function *deleg_with_var_coll_aparam_ren* from the specification of Accept below and we also use the function *has_def_method* in place of *has_impl_method*.

**value**
  has_Accept_defined : Wf_Design_Renaming → **Bool**
  has_Accept_defined(dr) ≡
    has_def_method(Element, Accept, dr) ∧
    unique_method(Element, Accept, dr) ∧
    one_image_ren_pars_in_design
      (Element, Accept, Visitor, visitor_param, dr)

Property 4 is analogous to property 5 of the Mediator pattern.

**value**
  exists_concrete_Element : Wf_Design_Renaming → **Bool**
  exists_concrete_Element(dr) ≡ exists_role(ConcreteElement, dr),

  is_concrete_Element : Wf_Design_Renaming → **Bool**

```
    is_concrete_Element(dr) ≡
      is_concrete(Element, ConcreteElement, dr)
```

The first clause of property 5 is analogous to property 3 of the Template Method pattern. The fact that all VisitConcreteElement methods have parameters of different types is specified by the function *differents_params*, while the other properties of the parameter are analogous to the properties of the parameter of the SetMemento method in the Memento pattern (see property 12, Section 2.6) and so are specified using the function *one_image_ren_pars_in_design*.

```
value
  has_VisitConcreteElement_defined : Wf_Design_Renaming → Bool
  has_VisitConcreteElement_defined(dr) ≡
    has_def_method(Visitor, VisitConcreteElement, dr) ∧
    has_all_def_method(Visitor, VisitConcreteElement, dr) ∧
    one_image_ren_pars_in_design
      (
        Visitor,
        VisitConcreteElement,
        ConcreteElement,
        concreteElement,
        dr
      ) ∧
    differents_params
      (Visitor, VisitConcreteElement, ConcreteElement, dr)
```

The sixth property is specified by directly equating the appropriate numbers of methods and classes, which are calculated using the functions *quantity_of_method* and *quantity_of_classes* respectively.

```
value
  equivalents_quantities : Wf_Design_Renaming → Bool
  equivalents_quantities(ds, r) ≡
    quantity_of_method(Visitor, VisitConcreteElement, r) =
    quantity_of_classes(ConcreteElement, r)
```

Property 7 is entirely analogous to property 4.

```
value
  exists_concrete_Visitor : Wf_Design_Renaming → Bool
  exists_concrete_Visitor(dr) ≡ exists_role(ConcreteVisitor, dr),
```

```
is_concrete_Visitor : Wf_Design_Renaming → Bool
is_concrete_Visitor(dr) ≡
   is_concrete(Visitor, ConcreteVisitor, dr)
```

Property 8 is analogous to property 5 of the Template Method pattern.

```
value
   all_VisitConcreteElement_impl : Wf_Design_Renaming → Bool
   all_VisitConcreteElement_impl(dr) ≡
      has_all_impl_method(ConcreteVisitor, VisitConcreteElement, dr)
```

Property 9 is analogous to property 12 of the Command pattern (see Section 2.8) and the specification can be simplified in the same way for the same reason.

```
value
   has_Operation_implemented : Wf_Design_Renaming → Bool
   has_Operation_implemented(dr) ≡
      has_impl_method(ConcreteElement, Operation, dr)
```

Property 10 is new and is defined using the function *visitor* which checks precisely the properties required.

```
value
   has_Accept_implemented : Wf_Design_Renaming → Bool
   has_Accept_implemented(dr) ≡
      visitor
         (
            ConcreteElement,
            Accept,
            visitor_param,
            Visitor,
            VisitConcreteElement,
            dr
         )
```

Property 11 is analogous to property 2 of the State pattern (see Section 2.3).

**value**
exists_one_ObjectStructure : Wf_Design_Renaming → **Bool**
exists_one_ObjectStructure(dr) ≡
exists_one(ObjectStructure, dr) ∧
is_concrete_class(ObjectStructure, dr)

Property 12 is analogous, up to the type and cardinality of the relation, to the first part of property 9 of the State pattern.

**value**
ObjectStructure_relation : Wf_Design_Renaming → **Bool**
ObjectStructure_relation(dr) ≡
has_assoc_aggr_reltype
(ObjectStructure, Element, AssAggr, G.many, dr)

Property 13 is also new and is defined using the function *ob_st_annotation* which checks precisely what is needed.

**value**
ObjectStructure_request_accept : Wf_Design_Renaming → **Bool**
ObjectStructure_request_accept(dr) ≡
ob_st_annotation
(ObjectStructure, Element, SendVisitor, Accept, dr)

As usual, we combine all the above properties to obtain the following function which checks whether a design represents a Command pattern:

**value**
is_visitor : Wf_Design_Renaming → **Bool**
is_visitor(dr) ≡
Visitor_hierarchy(dr) ∧
Element_hierarchy(dr) ∧
exists_one_ObjectStructure(dr) ∧
exists_concrete_Visitor(dr) ∧
exists_concrete_Element(dr) ∧
is_concrete_Visitor(dr) ∧
is_concrete_Element(dr) ∧
equivalents_quantities(dr) ∧
has_VisitConcreteElement_defined(dr) ∧
all_VisitConcreteElement_impl(dr) ∧

has_Accept_defined(dr) $\wedge$
has_Operation_implemented(dr) $\wedge$
has_Accept_implemented(dr) $\wedge$
ObjectStructure_relation(dr) $\wedge$ ObjectStructure_request_accept(dr)

# 3 An Example: Instantiation of the State Pattern

In this section we give an example of how an object-oriented design is represented in our model and how we relate this to a pattern using the renaming map. As the basis for this, we use the example which is used in [5] to illustrate the motivation and sample code of the State pattern.

This example is a model of a TCP network connection. This connection can be in one of several states – closed, established, listening, etc. – and different operations can be applied to these states to manipulate the connection.

The OMT-extended diagram of this design, where we only include classes representing the three states mentioned above, is shown in Figure 22.

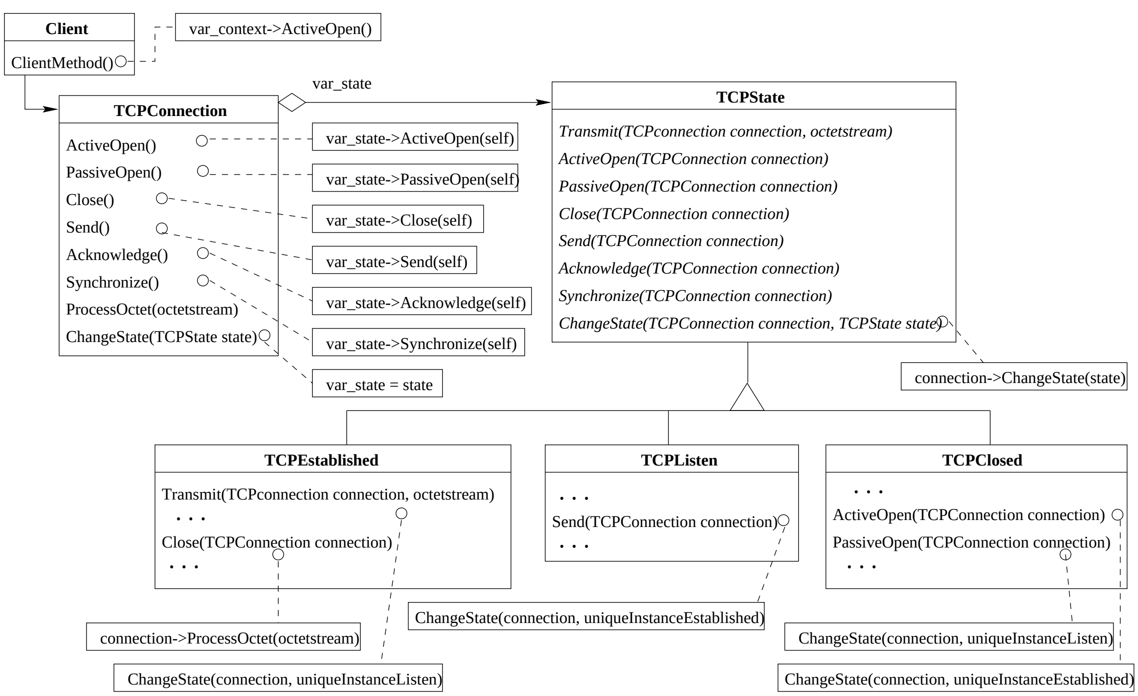


Figure 22: Design for a TCP Network Connection

We give only a representative sample of the specification of the design here, defining only the class TCPConnection and the relations in detail. The complete specification can be found in

Appendix A. The reader is also referred to [4] for the definitions of the types and values used in the specification below.

We begin by defining RSL constants which represent the names of the classes, methods, state variables and parameters which are used in the design. Those used in the class TCPConnection and the relations are:

```
value
  TCPConnection : G.Class_Name,
  TCPState : G.Class_Name,
  TCPEstablished : G.Class_Name,
  Client : G.Class_Name,
  ActiveOpen : G.Method_Name,
  PassiveOpen : G.Method_Name,
  Close : G.Method_Name,
  Send : G.Method_Name,
  Acknowledge : G.Method_Name,
  Synchronize : G.Method_Name,
  ProcessOctet : G.Method_Name,
  ChangeState : G.Method_Name,
  var_state : G.Variable_Name,
  var_context : G.Variable_Name,
  state : G.Variable_Name,
  octetstream : G.Variable_Name
```

Next we define the other parts of the methods – their bodies, results and parameters – and the collection of all methods in the class.

Although there are eight methods in the class TCPConnection, the forms of ActiveOpen, PassiveOpen, Close, Send, Acknowledge and Synchronize are essentially the same: each has no parameters, returns no result, causes no variable changes, and has a body which consists of a single invocation to the var_state variable of the corresponding method (i.e. the method with the same name) in the TCPState class, the parameter of each invocation being self. The specifications of these six methods are therefore all identical up to the names involved. Therefore we only show the specification of one of them, ActiveOpen, here together with the specifications of ProcessOctet and ChangeState.

We first define constants representing the bodies of the methods.

Because there are many cases in which different methods in the design have essentially the same structure as, for example, with the six methods described above, we introduce generic parameterised functions to represent these common structures and then define the individual methods in terms of these. An additional advantage of this approach is that the generic functions are likely to be reusable across many different designs.

We therefore begin by defining the function *one_inv_meth_body* which descibes in parameterised form the bodies of the first six methods in TCPConnection. This function then basically describes the body of any method in the design which consists of a single invocation to a given variable of a given method, the invocation having a single given parameter and the method involving no variable changes. Note that the invoked method and its parameter form an *actual signature* (see [4]) in the specification. Then the body of the ActiveOpen method is represented by a constant, *meth_body_AOctn*, which is constructed by instantiating the function *one_inv_meth_body* appropriately, in this case with the values var_state, ActiveOpen and self.

**value**
one_inv_meth_body :
G.Variable_Name × G.Method_Name × G.Wf_Variable_Name →
M.Method_Body
one_inv_meth_body(v, m, p) ≡
M.implemented
([ ], ⟨M.mk_Invocation(v, M.mk_Actual_Signature(m, ⟨p⟩))⟩),

meth_body_AOctn : M.Method_Body =
one_inv_meth_body(var_state, ActiveOpen, G.self),

The ProcessOctet and ChangeState methods are treated similarly. The first of these, like several other methods in the design, has no explicit body, so we introduce a generic constant *empty_method_body* to represent the body of all such methods. The second simply assigns its parameter to a particular state variable, so its body is empty apart from a single variable change which represents this assignment. This type of body is modelled generically using the function *assign_param_meth_body* and the body of the ChangeState method is again obtained by instantiating this function appropriately, in this case with the variables var_state and state.

**value**
empty_method_body : M.Method_Body = M.implemented([ ], ⟨⟩),

assign_param_meth_body :
G.Variable_Name × G.Wf_Variable_Name → M.Method_Body
assign_param_meth_body(v, p) ≡
M.implemented([ {v} ↦ M.Request_or_Var_from_Variable(p) ], ⟨⟩),

meth_body_ChgSt : M.Method_Body =
assign_param_meth_body(var_state, state)

Having defined the bodies of the methods, we now proceed to define the methods as a whole.

Again there are similarities in the structure of the methods: the first six methods in the TCPConnection class all have no parameters and no result, though they have different bodies; the method ProcessOctet has a single untyped parameter and no result; and the method ChangeState, in common with the majority of the methods in the other classes, has a single typed parameter and no result. We therefore introduce the two generic functions *method_with_body* and *method_with_body_param* to describe each of these forms in an appropriately parameterised way.

**value**
  method_with_body : M.Method_Body → M.Method
  method_with_body(b) ≡ M.mk_Method(⟨⟩, {}, b),

  method_with_body_param :
    M.Method_Body × G.Wf_Formal_Parameters → M.Method
  method_with_body_param(b, p) ≡ M.mk_Method(p, {}, b)

Then the specifications of the individual methods in the class TCPConnection are obtained by appropriately instantiating these generic functions, and the collection of all methods in the class, which is represented by the RSL constant *Ctn_Class_Methods*, is formed by constructing a map from each method name to the appropriate method. However, the methods constructed by these generic functions do not necessarily satisfy the well-formedness condition *is_wf_method* (the result type of the functions is *Method* not *Wf_Method*). Similarly, the collection of methods must satisfy the well-formedness condition *is_wf_class_method*. We must therefore check that these conditions are satisfied in order to be certain that the design is well-formed and the definition below is correctly typed.

**value**
  Ctn_Class_Methods : M.Class_Method =
    [
      ActiveOpen ↦ method_with_body(meth_body_AOctn),
      PassiveOpen ↦ method_with_body(meth_body_POctn),
      Close ↦ method_with_body(meth_body_Cctn),
      Send ↦ method_with_body(meth_body_Sctn),
      Acknowledge ↦ method_with_body(meth_body_Akctn),
      Synchronize ↦ method_with_body(meth_body_Syctn),
      ProcessOctet ↦
        method_with_body_param
          (empty_method_body, ⟨G.var(octetstream)⟩),
      ChangeState ↦
        method_with_body_param
          (meth_body_ChgSt, ⟨G.paramTyped(state, TCPState)⟩)
    ]

The sets of methods for the other classes are defined similarly.

The next step is to incorporate the definitions of the methods in the class into a definition of the class as a whole. For this we need to additionally define the class state and its type.

In the design, the class TCPConnection has a single state variable var_state and is a concrete class. The specification of this class, which we must again check for well-formedness (the function *is_wf_class*) is therefore:

```
value
   Ctn_Class : C.Wf_Class =
      C.mk_Design_Class({var_state}, Ctn_Class_Methods, G.concrete)
```

Next we turn to the relations in the design. There are in fact one aggregation relation, one association relation and three inheritance relations (one between TCPState and each of its subclasses) included. Here we only show the specification of one of the inheritance relations because the others are entirely analogous up to the names of the classes involved.

The aggregation and association relations are both one-one, so their specifications (the constants *agg_rel* and *ass_rel* respectively) are similar apart from their types and the names of the classes and variables involved. The inheritance relations are simply specified as inheritance relations between the appropriate pair of classes. Each must of course be shown to satisfy the well-formedness condition *wf_relation*.

```
value
   agg_rel : R.Wf_Relation =
      R.mk_Design_Relation
         (
            R.aggregation(R.mk_Ref(var_state, G.one, G.one)),
            TCPConnection,
            TCPState
         ),

   ass_rel : R.Wf_Relation =
      R.mk_Design_Relation
         (
            R.association(R.mk_Ref(var_context, G.one, G.one)),
            Client,
            TCPConnection
         ),

   inh1_rel : R.Wf_Relation =
      R.mk_Design_Relation(R.inheritance, TCPState, TCPEstablished)
```

The other classes and relations in the design are specified in a similar way, then the specification of the design as a whole is obtained by combining them together. To do this, we construct a map which associates each class name in the design with its definition and a set containing all the relations in the design. The design as a whole is then represented by the pair constructed from these two components. Checking the remaining well-formedness conditions then ensures that the design as a whole is well-formed.

**value**
  Class_Map : C.Classes =
    [
      Client ↦ Cli_Class,
      TCPConnection ↦ Ctn_Class,
      TCPState ↦ Sta_Class,
      TCPEstablished ↦ Est_Class,
      TCPListen ↦ Lis_Class,
      TCPClosed ↦ Clo_Class
    ],

  Rel_set : R.Wf_Relation-**set** =
    {agg_rel, inh1_rel, inh2_rel, inh3_rel, ass_rel},

  State_DS : DS.Wf_Design_Structure = (Class_Map, Rel_set)

This completes the specification of the design and we must now link the design to the pattern by defining a renaming mapping from the names of the classes, methods, state variables and parameters in the design to the corresponding entities which represent their roles in the pattern (see Figure 4 in Section 2.3). Again we concentrate on the class TCPConnection here. The full definition of the renaming map can also be found in Appendix A.

The class TCPConnection corresponds to the Context class in the pattern, and the first six methods (ActiveOpen, PassiveOpen, Close, Send, Acknowledge, and Synchronize) in TCPConnection all correspond to the Request operation in the pattern. Thus, in this example there are many elements of the design which play a single role in the pattern.

Since all the above methods play the same role in the pattern and have no explicit parameters, they all have the same renaming. We therefore simplify our specification by introducing a constant *Ctn_req_mtd* which represents this renaming. Then we construct a renaming map *Ctn_mtd* for the methods (and their parameters) by mapping each of the methods at the design level to this constant.

Note that the methods ProcessOctet and ChangeState have no counterparts in the pattern so are simply omitted from the method renaming map.

**value**

Ctn_req_mtd : Method_Renaming = mk_Method_Renaming(S.Request, [ ]),

Ctn_mtd : Method_and_Parameter_Renaming =
[
ActiveOpen ↦ Ctn_req_mtd,
PassiveOpen ↦ Ctn_req_mtd,
Close ↦ Ctn_req_mtd,
Send ↦ Ctn_req_mtd,
Acknowledge ↦ Ctn_req_mtd,
Synchronize ↦ Ctn_req_mtd
]

We similarly build a variable renaming map to associate the state variables in the TCPConnection class with those in the Context class. This is then combined with the method renaming to yield the renaming for the whole class.

**value**
Ctn_vbles : VariableRenaming = [ var_state ↦ S.state ],

Ctn_Class_Renaming : ClassRenaming =
mk_ClassRenaming(S.Context, Ctn_mtd, Ctn_vbles)

We follow the same procedure for the other classes in the design to obtain the renaming for the whole design, which simply associates the names of the classes in the design with the appropriate class renaming. Note that each design class plays a single role in the pattern so there is only a single class renaming for each design class. Again, we must check that the well-formedness condition *is_wf_Renaming* is satisfied.

**value**
State_Renaming : Renaming =
[
TCPConnection ↦ {Ctn_Class_Renaming},
TCPState ↦ {Sta_Class_Renaming},
TCPEstablished ↦ {Con_Class_Renaming},
TCPListen ↦ {Con_Class_Renaming},
TCPClosed ↦ {Con_Class_Renaming},
Client ↦ {Cli_Class_Renaming}
]

The final step is to combine the specifications of the design and the renaming and to check that these together satisfy the well-formedness condition *is_wf_design_renaming*.

**value**
State_Pat_Ren : Design_Renaming = (State_DS, State_Renaming)

This value is then used as input to the function *is_state_pattern* defined in Section 2.3 to check whether or not the TCP network connection design is an instance of the State pattern.

# 4 Classifying the Behavioural Patterns

In Section 2 we have described nine of the behavioural patterns and formally specified their properties. In this section we present a possible classification of the behavioural patterns based on common aspects identified during our analysis and specification.

Behavioural patterns describe not just patterns of objects or classes but also the patterns of communication between them [5]. In addition, they characterize the way in which classes and objects interact and distribute responsibilities, and each encapsulates one aspect that changes frequently in a program.

We classify the behavioural patterns considering the ways in which objects of the classes are interconnected and communicate with each other, and also how they provide object evolution (how the action that can be performed on an object or a set of objects changes or seems to change). We identify two main categories of behavioural patterns – communication between peer objects, and variation in behaviour encapsulated in and altered by a context – and a number of subcategories of these.

## 4.1 Communication between peer objects

The participant classes in the patterns belonging to this group are tightly coupled. The behaviour of the classes is defined by a set of requests that have to be performed in some methods in order for the participants of the pattern to correctly carry out their responsibilities. This group contains most of the patterns describing a group of peer objects that cooperate to perform a task that no single object can carry out by itself [5]. Different subgroups reflect the three different basic mechanisms by which this cooperation is achieved: through instance variables, objects, and intermediaries.

### Communication using instance variables

The classes in the patterns in this category collaborate by interchanging values of instances variables. The Memento and Observer patterns fall into this category. Both of these patterns have classes in which there are operations for accessing and updating their instances variables, the GetState and SetState methods respectively.

In the Memento pattern, the Memento class stores part of the state of the Originator class, and the state of the Memento class is used to restore the state of the Originator class if necessary. The GetState method is used to obtain the relevant state of the originator when a new memento is created and the SetState method is used to restore the originator's state to that of the memento (see Figure 10).

In the Observer pattern, the ConcreteObserver classes have a copy of the relevant parts of the state of the ConcreteSubject class they observe. The state of a subject is changed by an invocation of a SetState method, then the subject informs its observers that it has been updated and the observers copy the subject's state to their own using the GetState methods (see Figure 13).

### Communication using objects

The classes in the patterns in this group communicate by passing an object as a parameter to a method invocation. The Memento, Command and Visitor patterns fall into this category.

In the Visitor pattern, the SendVisitor method in the ObjectStructure class invokes the Accept method on the elements of the structure, passing the visitor as a parameter to this invocation. Then, when the Accept operation is executed, the appropriate VisitConcreteElement method is invoked on that visitor, the element passing itself as a parameter in this case (see Figures 21 and 20).

Similarly, in the Memento pattern, the caretaker passes its memento to the originator as a parameter of the SetMemento method when the state of the originator needs to be restored (see Figures 10 and 9).

Finally, in the Command pattern the client uses the ClientMethod to create a command and specifies the receiver of that command by passing the receiver as a parameter to the instantiation. Then the new command is itself passed as a parameter to an invocation of the StoreCommand method in the invoker (see Figures 17 and 15).

Note that the Observer pattern is not included in this category even though it uses an object as a parameter in the Attach, Detach and SetState methods. This is because these methods are not explicitly concerned with the collaborations of the pattern.

### Communication through intermediaries

Objects from the patterns in this category do not communicate directly. Rather they communicate indirectly through intemediaries. The Mediator and Chain of Responsibility patterns belong to this category.

Figure 23 shows a typical object structure at run-time of a chain of responsibility. The client issues a request to the first handler in the chain, and the request is passed along the chain until it reaches an object which can perform the requested action.

Similarly, in the Mediator pattern, mediator objects act as intermediaries between colleague objects: colleagues do not interact with each other directly, but instead pass requests to other colleagues to the mediator which then forwards them appropriately. A typical object structure for this interaction is shown in Figue 24.

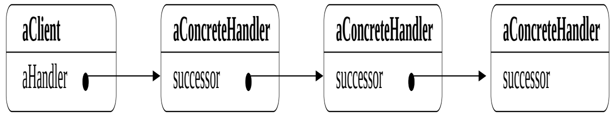


Figure 23: Object Structure in Chain of Responsibility Pattern

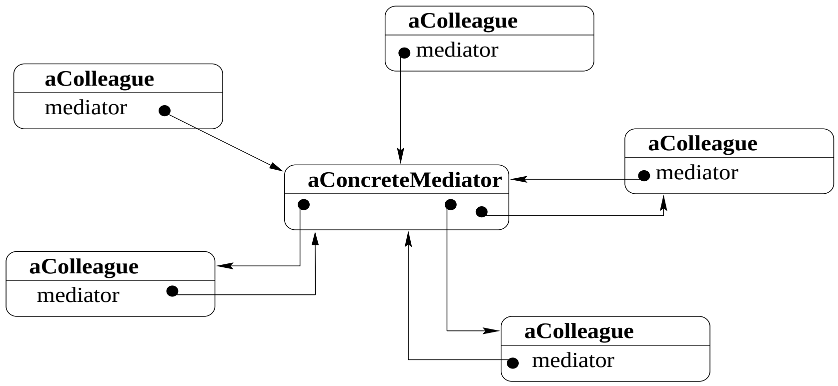


Figure 24: Object Structure in Mediator Pattern

## 4.2 Variation encapsulated in and altered by a context

Although all the behavioural patterns in the GoF catalogue encapsulate some aspect that varies in a program, the patterns in this category do not reflect a specific communication between peer objects. Instead, they provide behavioural variation in a particular context.

### Object changes behaviour within a particular context

One of the simplest forms of object evolution is modelled in the structure of the State and Strategy patterns in which there is one static aggregation between a context class and an abstract class and the concrete subclasses of the abstract class represent the behavioural variations needed in the context class.

In the State pattern, the various subclasses of the abstract State class will have different behaviour and the appropriate subclass, and hence the behaviour that is appropriate for the Context class is indicated by the state variable state (see Figure 4).

In the Strategy pattern, alternative implementations of a method are provided by the various subclasses of the Strategy class and the Context class similarly selects the appropriate implementation by referring to its strategy state variable (see Figure 5).

A more detailed explanation of how the State and Strategy patterns allow evolution of object behaviour using a context relation is given in [10].

### Behavioural class patterns and patterns used for implementation

The GoF catalogue [5] classifies patterns according to whether they are primarily concerned with classes (*class patterns*) or objects (*object patterns*). The behavioural class patterns basically use inheritance to distribute behaviour between classes [5]. The Template Method and Interpreter patterns form this category. Note, however, that some authors have considered that Interpreter is not application domain independent since it is used specifically to model language grammars.

Patterns that are used in implementation are intended to solve or improve particular problems in implemenation. The Template Method and Iterator patterns fall into this category. The Template Method pattern deals with the implementation of steps of an algorithm, while the Iterator pattern deals with access to and traversal of the components of an aggregate object.

## 5 Conclusion

We have presented an analysis of the essential components and properties of nine of the eleven GoF behavioural patterns and we have shown how the generic formal model of object-oriented design described in [4] can be used to formally specify these properties, and hence to formally specify functions which can check whether or not a particular design fragment matches a particular pattern. In this analysis we have included not only those properties of the patterns which are explicitly stated in the GoF catalogue [5] but also those which are implicit in the description, intent, motivation, and so on of the pattern as well as those which are implicit in the names of the classes, methods and variables used, and on this basis we have presented extended versions of the pattern structure which make these additional implicit properties explicit.

All the properties of the patterns are defined in terms of generic properties of the design together with a renaming mapping which links the design and pattern components appropriately. We have seen in this work how many of these generic properties are shared by several patterns, and we therefore expect that the same properties could form the basis for the description of other object-oriented patterns. We have also seen that the similarity of patterns like State and Strategy, which have analogous structures but which have different areas of applicability, can be captured very concisely in our formal model by using a formal renaming.

Many of the behavioural patterns admit one or more variants, including simplifications, such as degenerate hierarchies, and extensions, such as undo operations in the Command pattern. We have largely ignored these in this current work, though we have in many cases pointed out where variants are possible and we have included one such variant, namely the extension of the Command pattern to include macro commands. We intend to consider these variants more fully in future work.

We have also restricted our attention to single patterns in isolation, although many of the behavioural patterns are in fact closely related and indeed the GoF catalogue includes several examples of using a behavioural pattern in combination with another patterns in order to obtain a refined solution. We believe, however, that the model presented in [4] could be extended with fairly limited modification to allow us to describe such combinations of patterns and we plan to investigate this.

Finally, we hope to be able to use our general model together with the specifications of the individual patterns as the basis for designing a software system which could support both the use of and the identifiation of patterns in object-oriented designs.

# A Specification of the TCP Network Connection

**object** S : STATE

**value**
TCPConnection : G.Class_Name,

TCPState : G.Class_Name,

TCPEstablished : G.Class_Name,

TCPListen : G.Class_Name,

TCPClosed : G.Class_Name,

Client : G.Class_Name,

ActiveOpen : G.Method_Name,

PassiveOpen : G.Method_Name,

Close : G.Method_Name,

Send : G.Method_Name,

Acknowledge : G.Method_Name,

Synchronize : G.Method_Name,

ProcessOctet : G.Method_Name,

ChangeState : G.Method_Name,

Transmit : G.Method_Name,

ClientMethod : G.Method_Name,

var_state : G.Variable_Name,

var_context : G.Variable_Name,

connection : G.Variable_Name,

state : G.Variable_Name,

octetstream : G.Variable_Name,

uniqueInstanceEstablished : G.Variable_Name,

uniqueInstanceListen : G.Variable_Name,

uniqueInstanceClosed : G.Variable_Name,

Ctn_vbles : VariableRenaming = [var_state ↦ S.state],

Ctn_req_mtd : Method_Renaming = mk_Method_Renaming(S.Request, [ ]),

Ctn_mtd : Method_and_Parameter_Renaming =
[
ActiveOpen ↦ Ctn_req_mtd,
PassiveOpen ↦ Ctn_req_mtd,
Close ↦ Ctn_req_mtd,
Send ↦ Ctn_req_mtd,
Acknowledge ↦ Ctn_req_mtd,
Synchronize ↦ Ctn_req_mtd
],

Ctn_Class_Renaming : ClassRenaming =
mk_ClassRenaming(S.Context, Ctn_mtd, Ctn_vbles),

Cli_Class_Renaming : ClassRenaming =
mk_ClassRenaming(S.Client, [ ], [ ]),

Sta_han_mtd : Method_Renaming = mk_Method_Renaming(S.Handle, [ ]),

Sta_mtd : Method_and_Parameter_Renaming =
[
Transmit ↦ Sta_han_mtd,

```
        ActiveOpen ↦ Sta_han_mtd,
        PassiveOpen ↦ Sta_han_mtd,
        Close ↦ Sta_han_mtd,
        Send ↦ Sta_han_mtd,
        Acknowledge ↦ Sta_han_mtd,
        Synchronize ↦ Sta_han_mtd
    ],

Sta_Class_Renaming : ClassRenaming =
    mk_ClassRenaming(S.State, Sta_mtd, [ ]),

Con_Class_Renaming : ClassRenaming =
    mk_ClassRenaming(S.ConcreteState, Sta_mtd, [ ]),

State_Renaming : Renaming =
    [
        TCPConnection ↦ {Ctn_Class_Renaming},
        TCPState ↦ {Sta_Class_Renaming},
        TCPEstablished ↦ {Con_Class_Renaming},
        TCPListen ↦ {Con_Class_Renaming},
        TCPClosed ↦ {Con_Class_Renaming},
        Client ↦ {Cli_Class_Renaming}
    ],

one_inv_meth_body :
    G.Variable_Name × G.Method_Name × G.Wf_Variable_Name →
        M.Method_Body
one_inv_meth_body(v, m, p) ≡
    M.implemented
        ([ ], ⟨M.mk_Invocation(v, M.mk_Actual_Signature(m, ⟨p⟩))⟩),

assign_param_meth_body :
    G.Variable_Name × G.Wf_Variable_Name → M.Method_Body
assign_param_meth_body(v, p) ≡
    M.implemented([ {v} ↦ M.Request_or_Var_from_Variable(p) ], ⟨⟩),

meth_body_AOctn : M.Method_Body =
    one_inv_meth_body(var_state, ActiveOpen, G.self),

meth_body_POctn : M.Method_Body =
    one_inv_meth_body(var_state, PassiveOpen, G.self),

meth_body_Cctn : M.Method_Body =
    one_inv_meth_body(var_state, Close, G.self),
```

meth_body_Sctn : M.Method_Body =
  one_inv_meth_body(var_state, Send, G.self),

meth_body_Akctn : M.Method_Body =
  one_inv_meth_body(var_state, Acknowledge, G.self),

meth_body_Syctn : M.Method_Body =
  one_inv_meth_body(var_state, Synchronize, G.self),

meth_body_ChgSt : M.Method_Body =
  assign_param_meth_body(var_state, state),

method_with_body : M.Method_Body → M.Method
method_with_body(b) ≡ M.mk_Method(⟨⟩, {}, b),

method_with_body_param :
  M.Method_Body × G.Wf_Formal_Parameters → M.Method
method_with_body_param(b, p) ≡ M.mk_Method(p, {}, b),

Ctn_Class_Methods : M.Class_Method =
  [
    ActiveOpen ↦ method_with_body(meth_body_AOctn),
    PassiveOpen ↦ method_with_body(meth_body_POctn),
    Close ↦ method_with_body(meth_body_Cctn),
    Send ↦ method_with_body(meth_body_Sctn),
    Acknowledge ↦ method_with_body(meth_body_Akctn),
    Synchronize ↦ method_with_body(meth_body_Syctn),
    ProcessOctet ↦
      method_with_body_param
        (empty_method_body, ⟨G.var(octetstream)⟩),
    ChangeState ↦
      method_with_body_param
        (meth_body_ChgSt, ⟨G.paramTyped(state, TCPState)⟩)
  ],

Ctn_Class : C.Wf_Class =
  C.mk_Design_Class({var_state}, Ctn_Class_Methods, G.concrete),

par_tran : G.Wf_Formal_Parameters =
  ⟨G.paramTyped(connection, TCPConnection), G.var(octetstream)⟩,

par_ctn : G.Wf_Formal_Parameters =
  ⟨G.paramTyped(connection, TCPConnection)⟩,

cha_ctn : G.Wf_Formal_Parameters =

```
⟨
    G.paramTyped(connection, TCPConnection),
    G.paramTyped(state, TCPState)
⟩,

meth_body_defined : M.Method_Body = M.defined,

empty_method_body : M.Method_Body = M.implemented([ ], ⟨⟩),

meth_body_CSctn : M.Method_Body =
  one_inv_meth_body(connection, ChangeState, state),

Sta_Class_Method : M.Class_Method =
  [
    Transmit ↦ method_with_body_param(meth_body_defined, par_tran),
    ActiveOpen ↦
      method_with_body_param(meth_body_defined, par_ctn),
    PassiveOpen ↦
      method_with_body_param(meth_body_defined, par_ctn),
    Close ↦ method_with_body_param(meth_body_defined, par_ctn),
    Synchronize ↦
      method_with_body_param(meth_body_defined, par_ctn),
    Acknowledge ↦
      method_with_body_param(meth_body_defined, par_ctn),
    Send ↦ method_with_body_param(meth_body_defined, par_ctn),
    ChangeState ↦ method_with_body_param(meth_body_CSctn, cha_ctn)
  ],

Sta_Class : C.Wf_Class =
  C.mk_Design_Class({}, Sta_Class_Method, G.abstract),

meth_body_TEst : M.Method_Body =
  one_inv_meth_body(connection, ProcessOctet, octetstream),

meth_body_to_Lis : M.Method_Body =
  M.implemented
    (
      [ ],
      ⟨
        M.mk_Invocation
          (
            G.self,
            M.mk_Actual_Signature
              (ChangeState, ⟨connection, uniqueInstanceListen⟩)
          )
```

```
            ⟩
        ),

    Est_Class_Method : M.Class_Method =
        [
            Transmit ↦ method_with_body_param(meth_body_TEst, par_tran),
            ActiveOpen ↦
                method_with_body_param(empty_method_body, par_ctn),
            PassiveOpen ↦
                method_with_body_param(empty_method_body, par_ctn),
            Close ↦ method_with_body_param(meth_body_to_Lis, par_ctn),
            Synchronize ↦
                method_with_body_param(empty_method_body, par_ctn),
            Acknowledge ↦
                method_with_body_param(empty_method_body, par_ctn),
            Send ↦ method_with_body_param(empty_method_body, par_ctn)
        ],

    meth_body_to_Est : M.Method_Body =
        M.implemented
            (
                [ ],
                ⟨
                    M.mk_Invocation
                        (
                            G.self,
                            M.mk_Actual_Signature
                                (
                                    ChangeState, ⟨connection, uniqueInstanceEstablished⟩
                                )
                        )
                ⟩
            ),

    Lis_Class_Method : M.Class_Method =
        [
            Transmit ↦ method_with_body_param(empty_method_body, par_tran),
            ActiveOpen ↦
                method_with_body_param(empty_method_body, par_ctn),
            PassiveOpen ↦
                method_with_body_param(empty_method_body, par_ctn),
            Close ↦ method_with_body_param(empty_method_body, par_ctn),
            Synchronize ↦
                method_with_body_param(empty_method_body, par_ctn),
            Acknowledge ↦
```

```
            method_with_body_param(empty_method_body, par_ctn),
        Send ↦ method_with_body_param(meth_body_to_Est, par_ctn)
    ],

Clo_Class_Method : M.Class_Method =
    [
        Transmit ↦ method_with_body_param(empty_method_body, par_tran),
        ActiveOpen ↦ method_with_body_param(meth_body_to_Est, par_ctn),
        PassiveOpen ↦
            method_with_body_param(meth_body_to_Lis, par_ctn),
        Close ↦ method_with_body_param(empty_method_body, par_ctn),
        Synchronize ↦
            method_with_body_param(empty_method_body, par_ctn),
        Acknowledge ↦
            method_with_body_param(empty_method_body, par_ctn),
        Send ↦ method_with_body_param(empty_method_body, par_ctn)
    ],

Est_Class : C.Wf_Class =
    C.mk_Design_Class({}, Est_Class_Method, G.concrete),

Lis_Class : C.Wf_Class =
    C.mk_Design_Class({}, Lis_Class_Method, G.concrete),

Clo_Class : C.Wf_Class =
    C.mk_Design_Class({}, Clo_Class_Method, G.concrete),

agg_rel : R.Wf_Relation =
    R.mk_Design_Relation
        (
            R.aggregation(R.mk_Ref(var_state, G.one, G.one)),
            TCPConnection,
            TCPState
        ),

ass_rel : R.Wf_Relation =
    R.mk_Design_Relation
        (
            R.association(R.mk_Ref(var_context, G.one, G.one)),
            Client,
            TCPConnection
        ),

inh1_rel : R.Wf_Relation =
    R.mk_Design_Relation(R.inheritance, TCPState, TCPEstablished),
```

```
inh2_rel : R.Wf_Relation =
   R.mk_Design_Relation(R.inheritance, TCPState, TCPListen),

inh3_rel : R.Wf_Relation =
   R.mk_Design_Relation(R.inheritance, TCPState, TCPClosed),

Rel_set : R.Wf_Relation-set =
   {agg_rel, inh1_rel, inh2_rel, inh3_rel, ass_rel},

Class_Map : C.Classes =
   [
      Client ↦ Cli_Class,
      TCPConnection ↦ Ctn_Class,
      TCPState ↦ Sta_Class,
      TCPEstablished ↦ Est_Class,
      TCPListen ↦ Lis_Class,
      TCPClosed ↦ Clo_Class
   ],

State_DS : DS.Wf_Design_Structure = (Class_Map, Rel_set),

State_Pat_Ren : Design_Renaming = (State_DS, State_Renaming),

meth_body_Cli : M.Method_Body =
   M.implemented
      (
         [ ],
         ⟨
            M.mk_Invocation
               (var_context, M.mk_Actual_Signature(ActiveOpen, ⟨⟩))
         ⟩
      ),

Cli_Class_Method : M.Class_Method =
   [ ClientMethod ↦ method_with_body(meth_body_Cli) ],

Cli_Class : C.Wf_Class =
   C.mk_Design_Class({}, Cli_Class_Method, G.concrete)
```

# References


[1] Kent Beck, James O. Coplien, Ron Crocker, Lutz Dominick, Gerard Meszaros, Frances Paulisch, and John Vlissides. Industrial Experience with Design Patterns. Technical report, First Class Software, AT&T, Motorola Inc, Siemens AG, Bell Northern Research, Siemens AG, and IBM Research. http://www1.bell-labs.com/user/cope/Patterns/ICSE96/icse.html.

[2] Paul Dyson and Bruce Anderson. *State Patterns, EuroPLoP '96 Writers Workshop, Pattern Languages of Program Design 3 (PLoPD3)*, chapter 9, pages 125–142. Addison-Wesley, 1998.

[3] Ammon H. Eden, Joseph Gil, and Amiram Yehudai. A Formal Language for Design Patterns. Technical report, The Department of Computer Science, School of Mathematics, Tel Aviv University of Israel. http://www.math.tau.ac.il/~eden/bibliography.html.

[4] Andres Flores, Luis Reynoso, and Richard Moore. A Formal Model of Object-Oriented Design and GoF Design Patterns. Technical Report 200, UNU/IIST, P.O. Box 3058, Macau, July 2000.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley Professional Computing Series. Addison Wesley, 1995.

[6] Ralph Johnson. Design Patterns in the Standard Java Libraries. In *Proceedings of the Asia Pacific Software Engineering Conference: Keynote Materials, Tutorial Notes*, pages 66–101, 1999.

[7] Henry Lieberman. There's more to menu systems than meets the screen. In *SIGGRAPH Computer Graphics*, pages 181–189, July 1995.

[8] Marco Meijers. Tool Support for Object-Oriented Design Patterns. Master's thesis, Department of Computer Science, Utrecht University, The Netherlands, August 1996.

[9] The RAISE Language Group. *The RAISE Specification Language.* BCS Practitioner Series. Prentice Hall, 1992.

[10] Linda M. Seiter. Design Patterns for Managing Evolution . Master's thesis, College of Computer Science, Northeastern University, September 1996. ftp://ftp.ccs.neu.edu/pub/people/lieber/theses/seiter/thesis.ps.

[11] Linda M. Seiter, Jens Palsberg, and Karl J. Lieberherr. Evolution of Object Behavior using Context Relations. In *SIGSOFT 96*, pages 46–57. ACM, 1996.

[12] John Vlissides. *Pattern Hatching: Design Patterns Applied.* Software Patterns Series. Addison-Wesley, 1998.

[13] John M. Vlissides and Mark A. Linton. Unidraw: A Framework for Building Domain-Specific Graphical Editors. *ACM Transactions on Information Systems*, 8(3):237–268, July 1990.