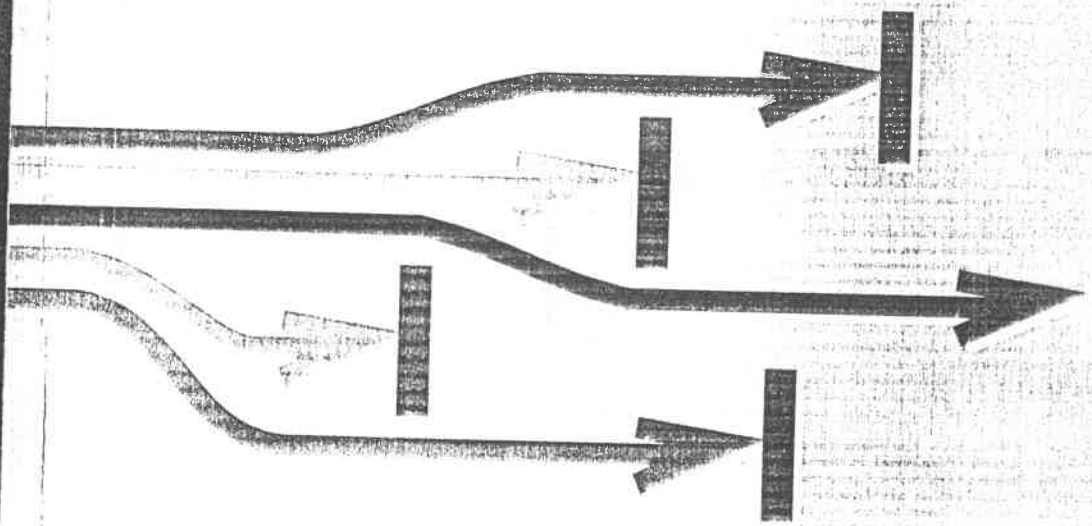
Hung Dang Van, Chris George, Tomasz Janowski
and Richard Moore (Eds)

Specification Case Studies in RAISE



Springer

FACIT

Study

ced Applications

Hung Dang Van, Chris George,
Tomasz Janowski and Richard Moore (Eds)

Specification Case Studies in RAISE

Springer

Formal Approaches to Computing and Information Technology –
FACIT - is the generic title of a series of books designed to fill the need for post-experience education in formal methods.

FACIT includes textbooks, case studies and reference works. The series brings together information for industrial and commercial practitioners who need to develop an in-depth understanding of the implications of formally-based approaches, for enhanced professional effectiveness.

Individual books under the ***FACIT*** heading inevitably span a wide spectrum of subjects but all address the same basic need: to develop an understanding of formal methods and apply them effectively for producing dependable software.

This volume presents twelve case studies that use RAISE – Rigorous Approach to Industrial Software Engineering – to construct, analyse, develop and apply formal specifications. The case studies cover a wide range of application areas including government finance, case-based reasoning, multi-language text processing, object-oriented design patterns, component-based software design and natural resource management. By illustrating the variety of uses of formal specifications, the case studies also raise questions about the creation, purpose and scope of formal models before they are built.

The complete specifications for all of the case studies, and the RAISE tools (both source code and executables) used to process them are available at:

http://www.iist.unu.edu/RAISE_Case_Studies/

This book will be of particular interest to software engineers, especially those responsible for the initial stages of requirements engineering and software architecture and design. It will also be of interest to academics and students on advanced formal methods courses.

ISBN 1-85233-359-6

www.springer-ny.com
www.springer.co.uk
www.springer.de

11. Object-Oriented Design Patterns

11.1 Introduction

Software *design patterns* offer designers a way of reusing proven solutions to particular aspects of design rather than having to start each new design from scratch – the patterns are generic and abstract and embody "best practice" solutions to design problems which recur in a range of different contexts [4].

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. [55]. The GoF catalogue is thus a description of the know-how of expert designers in problems appearing in various different domains.

The patterns in the GoF catalogue are described using a consistent format which is based on an extension of the general object-oriented design technique OMT (Object Modelling Technique [123]). A graphical notation is used to represent the main constituents of the pattern – classes, methods, and relationships between classes – and this is supplemented with natural language descriptions of the intent and motivation of the pattern and the roles and responsibilities of its constituents. In addition, examples of the use of the patterns, in the form of both designs and sample code, are included. This notation has in fact effectively been adopted as the standard way of presenting software design patterns.

This form of presentation gives a very good intuitive picture of the patterns, but it is not sufficiently precise to allow a designer to demonstrate conclusively that a particular problem matches a particular pattern or that a proposed solution is consistent with a particular pattern. Moreover, it also makes it difficult to be certain that the patterns themselves are meaningful and contain no inconsistencies. Indeed, in some cases the descriptions of the patterns are intentionally left loose and incomplete to ensure that they are applicable in as wide a range of applications as possible, which can make it difficult for designers to be sure that they have interpreted and understood the patterns correctly. A formal model of patterns can help to alleviate these problems.

[1] Gang of Four

In this chapter we present a formal model of a generic object-oriented design based upon the extended OMT notation, and show how designs can be formally linked with patterns in this model. Finally, we show how the properties of individual patterns can be specified in the model, thus giving a basis for formally checking whether a given design and a given pattern are consistent with each other.

11.2 A Formal Model of Object-Oriented Design

Both a general object oriented design in OMT and a description of a design pattern in the extended OMT are based around various kinds of diagram, the most important of which are class diagrams and interaction diagrams. Class diagrams show the different classes that make up the design together with their instance variables and methods, and also show the relationships between different classes. Interaction diagrams represent sequences of method calls and thus partly define the functionality of particular methods. Examples in the extended OMT notation of a class diagram and an interaction diagram, which in fact represent respectively the structure and collaborations of the Command pattern (see [55]), are shown in Figures 11.1 and 11.2.

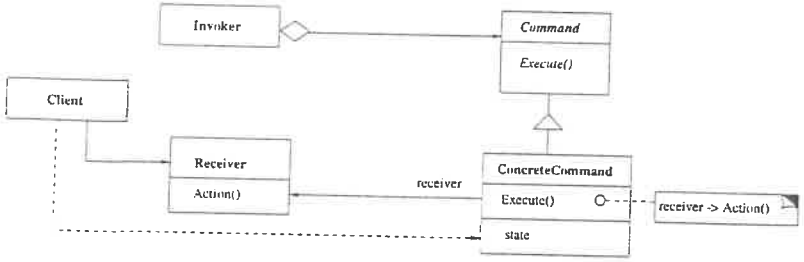


Fig. 11.1. Class diagram: Command pattern structure.

We begin by describing the main elements and properties informally in Section 11.2.1, then go on to develop a formal specification of these in Section 11.2.2.

11.2.1 The Extended OMT Notation

In the class diagram classes are depicted as rectangles containing the name of the class in bold face type at the top and below this the signatures (i.e. the names and the parameters) of the methods of the class followed by its instance variables. Every class in the design must have a unique name.

Classes and methods may be either *abstract* or *concrete*, an abstract class being one from which no instances (objects) can be created and an abstract

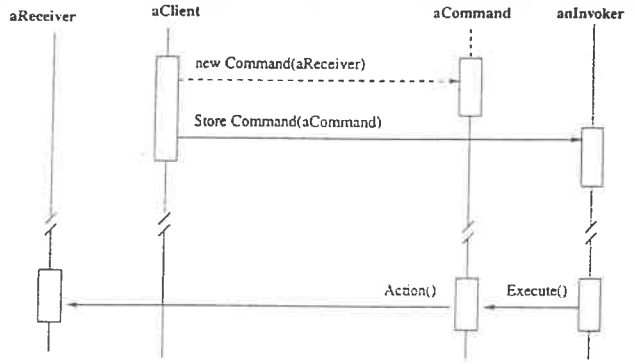


Fig. 11.2. Interaction diagram: collaborations of the Command pattern.

method one which cannot be executed (perhaps because the method is only completely defined in subclasses). An abstract class or method is indicated in the class diagram by writing its name in italic script instead of upright script.

Concrete methods, which can be executed, may additionally have *annotations* in the class diagram. These annotations indicate what actions the method should perform and they appear within rectangles with a "folded" corner which are attached to a method within the class description rectangle by a dashed line ending in a small circle.

Thus, for example, the class diagram in Figure 11.1 indicates that the class ConcreteCommand is a concrete class which contains a concrete method called Execute and a state variable called state, while the class Command is an abstract class in which the method called Execute is abstract. In addition, the annotation attached to the Execute method in the ConcreteCommand class indicates that the action of this method is to invoke the Action method on the variable called receiver.

The relationships in the class diagram indicate connections or communications between classes. They are represented as lines linking classes, four different kinds of lines being used to distinguish four different types of relations as follows:

`inheritance`: drawn as a solid line with a triangle in the middle and indicating that one class is a subclass of another. The base of the triangle lies towards the subclass. The inheritance relation between the Command class and the ConcreteCommand class in Figure 11.1 thus indicates that the ConcreteCommand class is a subclass of the Command class.

`instantiation`: drawn as a dashed line with an arrowhead on one end and indicating that one class creates objects belonging to another class. The arrowhead points to the class from which the objects are created. The instantiation relation between the Client class and the ConcreteCommand

class in the Command pattern thus indicates that the Client class creates ConcreteCommand objects.

`aggregation`: drawn as a solid line with a diamond on one end and an arrowhead on the other and indicating that objects of one class are constituent parts (sub-objects) of objects of another class. The arrowhead points towards the class of the sub-object. The aggregation relation between the Invoker class and the Command class in Figure 11.1 thus indicates that the Invoker class has a sub-object belonging to the Command class.

`association`: drawn as a solid line with an arrowhead on one end and indicating that one class communicates with another, generally by invoking methods of the second class. The arrowhead points in the direction of the communication. In the class diagram in Figure 11.1 there are association relations between the Client and the Receiver classes and between the ConcreteCommand and the Receiver classes corresponding to the invocation of methods in the Receiver class by methods in the Client class and the ConcreteCommand class.

Association and aggregation relations also have an associated *arity* and may additionally have an associated name. The arity indicates the number of objects participating in the relation, and may be either one or many according to whether each object of one class communicates with or is composed of a single object or a collection of objects of the other class. Relations of arity many are indicated by adding a solid black circle to the front of the arrowhead, as shown in the aggregation relation between the MacroCommand and the Command classes in Figure 11.3. Names of relations, which generally correspond to state variables (although the state variables may not be explicitly shown), are written above the line representing the relation as in the name commands of the same aggregation relation.

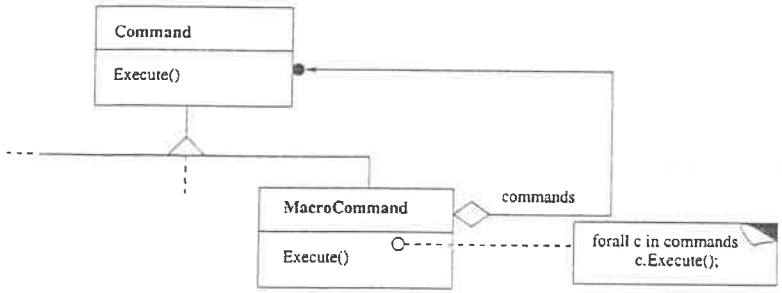


Fig. 11.3. The structure of the Macro Command.

The interaction diagram in Figure 11.2 gives details about the interactions between the Client, Invoker, Receiver and ConcreteCommand classes. It shows that the Client first creates a new instance of the ConcreteCommand class and

instantiates the receiver instance variable in that instance, then it passes this concrete command to the Invoker by invoking its StoreCommand method with the concrete command as its parameter.

11.2.2 Formalising the Components of the Model

The basic building blocks of design which are used in the class diagram are classes, methods, state variables, and relations. These are distinguished by their names, which we specify as sort types. However, the names of relations are in fact the same as the names of the state variables with which they are associated, so we only introduce types representing class names, method names, and variable names:

type
Class_Name, Method_Name, Variable_Name

We deal with each of the four elements separately.

Methods. The definition of a method in the extended OMT notation consists in general of two parts – its *signature*, which comprises its name, its input parameters and its result and which is written inside the rectangle representing the class to which the method belongs, and its (optional) *body*, which gives an indication of the actions that the method performs and which takes the form of a list of interactions appearing as an annotation in a separate rectangle which is linked to the signature.

In fact there are essentially only two different kinds of interactions that appear in the method body: invocations and instantiations.

An invocation represents an interaction in which objects of one class request objects of another (possibly the same) class to perform some action by executing some method. In the case where the two classes are different (see for example the description of the body of the Execute method in the ConcreteCommand class in the Command pattern illustrated in Figure 11.1), this corresponds to a communication between two classes and there will be either an association or an aggregation relation which links the two classes and which represents that communication. In addition, the name of the relation is the name of a variable which represents the receiver of the invocation (the variable receiver in the Execute method).

In the case where the two classes are the same, the invocation could be to a method in the same class or to a method in a superclass of that class. In general, however, there are no specific variables in the design which correspond to the receivers of these invocations. so we introduce two reserved variable names `self` and `super`, which respectively represent the receivers of invocations to the same class and to a superclass of the current class (cf. the variables of the same name in Smalltalk). The fact that these are different values is enforced by an axiom.

```
value
  self, super : Variable_Name

axiom
  self ≠ super
```

An invocation then consists of a variable name, which may be a variable defined in the design or `self` or `super`, together with the name of the method being invoked and the actual parameters passed to that method.

The actual parameters are basically just a list of variables, though they may not include the reserved variable `super`. We therefore introduce the subtype `Wf_Variable_Name` of `Variable_Name` whose defining predicate `wf_vble_name` excludes the value `super`, then use this subtype to define the type `Actual_Parameters` which represents the parameters of an invocation:

```
type
  Wf_Variable_Name = {| v : Variable_Name • wf_vble_name(v) |},
  Actual_Parameters = Wf_Variable_Name*

value
  wf_vble_name : Variable_Name → Bool
  wf_vble_name(v) ≡ v ≠ super
```

Next, we define the record type `Actual_Signature` to represent the name of the method being invoked together with its actual parameters, and an invocation as a whole is then represented by the record type `Invocation` which consists of the name of the variable representing the receiver of the request plus the appropriate signature:

```
type
  Actual_Signature ::
    meth_name : Method_Name  a_params : Actual_Parameters,
  Invocation ::
    call_vble : Variable_Name  call_sig : Actual_Signature
```

An instantiation represents an interaction in which one class requests another class to create a new object. In most object-oriented programming languages there are essentially two ways in which this can be done: either the class might create a "default" instance of itself and then set the state variables of this instance appropriately using other methods; or the class may have other local creation methods which have parameters which are used to create customised instances directly. We cover both of these situations in our model by representing an instantiation as the type `Instantiation` which consists of the name of the class to be instantiated together with a possibly empty list of actual parameters to be used by the instantiation method.


```
type
  Instantiation ::
    class_name : Class_Name  a_params : Actual_Parameters
```

A design may also indicate the results returned by methods and assignments to variables within the bodies of the methods, including assignments to both state variables and local (dummy) variables.

In an actual implementation, the result of a method is likely to be a tuple or a list of variables, but we model this abstractly and use just a set of variables since this is sufficient to capture all the necessary properties. However, it is clear that the result cannot include the variable `super`, so we use the type `Wf_Variable_Name` to define the type `Result`.

```
type
  Variables = Wf_Variable_Name-set,
  Result = Variables
```

Assignments come in two different forms: one in which the result of some *request* (i.e. instantiation, invocation, etc.) is assigned to one or more variables, generally so that it can be used as parameters or receivers of subsequent actions; and one where the parameters of the method are assigned to variables, generally state variables. We model the first of these as a mapping from sets of variables to requests, where requests are modelled using the union type `Request`, and the second as a mapping from sets of variables to sets of variables.

In both cases the domain value of the mapping should not include the variables `self` or `super` since it doesn't make sense to assign values to those variables, and in the second case the range value of the mapping should also not include the variable `super` but it may include the variable `self`. We therefore introduce another subtype (`Wf_Vble_Name`) of `Variable_Name` whose defining predicate `not_self_super` excludes these two values. Then both kinds of assignment are modelled by the map type `Variable_Change`, which maps sets of variables to either requests or sets of variables (the union type `Request_or_Var`) and which is also defined as a subtype, its defining predicate `is_wf_vchange` requiring that we do not make an assignment to the empty set of variables. Additional constraints on this "variable change map" are defined below as part of the well-formedness condition on a method as a whole.

```
type
  Wf_Vble_Name = {| v : Variable_Name • not_self_super(v) |},
  Request = Invocation | Instantiation | _ ,
  Request_or_Var = Request | Variables,
  Variable_Change =
    {| m : Wf_Vble_Name-set ~m~ Request_or_Var • is_wf_vchange(m) |}
```

```
value
  not_self_super : Variable_Name → Bool
  not_self_super(v) ≡ v ≠ self ∧ v ≠ super,

  is_wf_vchange : (Wf_Vble_Name-set ~m~ Request_or_Var) → Bool
  is_wf_vchange(m) ≡ {} ∉ dom m
```

As explained in Section 11.2.1, methods may be abstract or concrete. In general, only the signature is defined for an abstract method, which means that the method cannot be executed, whereas a concrete method is defined completely and can be executed. Abstract methods are commonly defined in superclasses, concrete definitions of the same method being given in the subclasses. However, in some cases it is useful to define some methods abstractly in a superclass even if they should not be executable or do not make sense in all subclasses. (See for example the Composite pattern in [55] in which the child management operations Add, Remove, and GetChild do not make sense at all in the Leaf classes although they are in principle available because they are inherited from the Component class.) In this case, the method should be concrete in all concrete subclasses but it is not executable. We therefore subdivide the classification of a concrete method into *error* methods and *implemented* methods in order to be able to distinguish these two cases.

Like an abstract method, an error method cannot be executed. Thus, both perform no actions and therefore have no body, so we represent their bodies simply by the constants `defined` and `error` respectively. An implemented method, on the other hand, must actually perform some action and so does have a body, though this could be empty since the actions of the method might not be specified in the design. In this case, the body consists of the requests and assignments contained in the annotation to the method, and we therefore model it using the `Variable_Change` mapping defined above together with a list of requests which represents the actions performed by the method in the order in which they are performed. The bodies of the three kinds of methods are then modelled using the variant type `Method_Body` as follows:

```
type
  Method_Body ==
    defined |
    error |
    implemented(variable_change: Variable_Change, request_list: Request*)
```

Finally, we come to the input parameters appearing in a method's signature, which we refer to as its formal parameters in order to distinguish them from the actual parameters used in invocations of the method. There are in fact two ways in which these formal parameters can be defined in the extended OMT notation: in the first case only the name of the parameter is given, while in the second the parameter name is accompanied by a class name which indicates the class of the object the parameter represents. Formal parameters may not include the values `self` or `super` since they represent generic "placeholders" for actual variables, and all variable names used in the formal parameters must be different so that the variables can be distinguished in the body of the method.

We use the variant type `Parameter` to represent the two different kinds of formal parameters, the variable names in both variants being represented by the type `Wf_Vble_Name` which automatically excludes the unwanted values `self` and `super`. Then the formal parameters of a method are described by the subtype `Wf_Formal_Parameters`, which is a list of parameters in which no two parameters have the same variable name.

The defining predicate `is_wf_formal_parameters` of the subtype, which represents the above consistency condition, is written in terms of the auxiliary function `type_parameter` which simply extracts the variable name from a parameter.

```
type
  Parameter ==
    var(Wf_Vble_Name) |
    paramTyped(paramName : Wf_Vble_Name, className : Class_Name),
  Wf_Formal_Parameters =
    {| p : Parameter* • is_wf_formal_parameters(p) |}

value
  is_wf_formal_parameters : Parameter* → Bool
  is_wf_formal_parameters(p) ≡
    (∀ i, j : Nat •
      {i, j} ⊆ inds p ∧
      type_parameter(p(i)) = type_parameter(p(j)) ⇒
      i = j),

  type_parameter : Parameter → Variable_Name
  type_parameter(p) ≡
    case p of var(v) → v, paramTyped(v, c) → v end
```

Putting all these various components together, we define a method as a whole using a record type which comprises the method's body, its result, and its formal parameters.

```
type
  Method ::
    f_params : Wf_Formal_Parameters
    meth_res : Result
    body : Method_Body
```

There are a number of consistency conditions which this record type must satisfy in order for it to be a valid representation of a method. These are:

`receiver_in_call_vble`: any request which appears in the range of the variable change mapping of an implemented method must appear in the list of requests comprising the body of that method;

`correct_list_ins`: the result of every instantiation in the body of an implemented method must be assigned to a variable in the variable change mapping and therefore must appear in the range of that mapping. This condition is in fact not strictly necessary but corresponds to a sort of "no waste" condition – the instantiation must return a new object and if this is not assigned to a variable it is just lost and there was therefore no point in creating it;

`is_correct_fparams`: the variable change mapping cannot make assignments to formal parameters of an implemented method;

`is_error_method`: an error method cannot return a result (i.e. its result must be an empty set of variables);

`correct_inst_assig`: if an instantiation appears in the variable change map, the set of variables to which it is assigned must contain exactly one variable (because the result of an instantiation is always a single variable; i.e. only one object is created);

`correct_vble_assig`: when the variable change map assigns one set of variables to another, the two sets must contain the same number of variables;

`correct_collel_assig`: if an invocation of the method `collectionelement`[2] appears in the variable change map, the set of variables to which it is assigned must contain exactly one variable (because its result is a single variable).

These seven functions are combined together in the function `is_wf_method`, and this is used as the defining predicate for the subtype `Wf_Method` of the type `Method`. This completes our definition of a method.

```
value
  is_wf_method : Method → Bool
  is_wf_method(m) ≡
    receiver_in_call_vble(m) ∧
    correct_list_ins(m) ∧
    is_correct_fparams(m) ∧
    is_error_method(m) ∧
    correct_inst_assig(m) ∧
    correct_vble_assig(m) ∧ correct_collel_assig(m)

type
  Wf_Method = {| m : Method • is_wf_method(m) |}
```

[2] This is one of a group of three reserved method names – `collectionadd`, `collectionremove` and `collectionelement` – which are introduced in order to model various operations on collections of objects which are not specified explicitly but which are implicit in the names of the methods used in a particular design or pattern, for instance the methods Attach and Detach in the Observer pattern in [55]. The method `collectionelement` returns one element from a collection.

The last step in this section is to define the collection of all methods in a class. This is done using the map type `Map_Methods`, which associates the name of each method with its definition. Using a finite map here automatically ensures that no two methods in the same class have the same name. At this level there is only one constraint, namely that the reserved method names `collectionadd`, `collectionremove` and `collectionelement` cannot be used as the names of methods in the design. This is expressed using the function `is_wf_class_method` and this is in turn used as the defining predicate of the subtype `Class_Method` which then represents the well-formed collection of all methods in a class.

```
type
  Map_Methods = Method_Name ~m~> Wf_Method,
  Class_Method = {| m : Map_Methods • is_wf_class_method(m) |}

value
  is_wf_class_method : Map_Methods → Bool
  is_wf_class_method(m) ≡
    collectionadd ∉ dom m ∧
    collectionremove ∉ dom m ∧ collectionelement ∉ dom m
```

State Variables. The state variables of a class can simply be described as a set of variables, except that the set cannot include the two reserved variables `self` and `super`. We therefore use the subtype `Wf_Vble_Name` of `Variable_Name` introduced above to define the type `State` which represents the state variables of a class:

```
type
  State = Wf_Vble_Name-set
```

Classes. Its state variables and methods are the main elements of a class, and these are modelled by the types `Class_Method` and `State` introduced above. However, as explained in Section 11.2.1, a class may be either `abstract` or `concrete`. We model this *class type* using the variant type `Class_Type`, and a class is then modelled as the record type `Design_Class` which is composed of the appropriate three components:

```
type
  Class_Type == abstract | concrete,
```

```
Design_Class ::
  class_state : State
  class_methods : Class_Method
  class_type : Class_Type
```

Again, the basic record type requires some constraints, though in this case only one, namely that state variables cannot be used as formal parameters of methods since this would lead to ambiguity when referring to the name in the body of the method. We define the function `is_wf_class` to capture this property and use this as the defining predicate for the subtype `Wf_Class` of well-formed classes. The function `method_in_class` used in the definition of `is_wf_class` simply checks that a given method belongs to a given class.

```
value
  is_wf_class : Design_Class → Bool
  is_wf_class(c) ≡
    (
      ∀ m : Wf_Method •
        method_in_class(m, c) ⇒
        class_state(c) ∩ set_f_params(f_params(m)) = {}
    ),

  method_in_class : Wf_Method × Design_Class → Bool
  method_in_class(m, c) ≡ m ∈ rng class_methods(c)

type
  Wf_Class = {| c : Design_Class • is_wf_class(c) |}
```

The collection of all classes in a design is then modelled analogously to the collection of all methods in a class – the map type `Classes` associates the name of each class with its definition. Again, using a finite map here automatically ensures that no two classes in the design have the same name.

```
type
  Classes = Class_Name ~m~> Wf_Class
```

Relations. A relation is basically a link between classes which has one of four distinct types (inheritance, association, aggregation, or instantiation) and which may also (in the case of association and aggregation relations only) have an arity. All relations except inheritance relations are binary and link a single *source* class to a single *sink* class. Inheritance relations are in general one-to-many, linking a superclass with an arbitrary number of subclasses. However, it is possible to view a single one-to-many inheritance relation which links one superclass to (say) n subclasses as n binary inheritance relations, each linking the superclass to one of the subclasses. This allows us to consider all relations as binary relations and thus simplify our model.

In the case of instantiation and (binary) inheritance relations, there can be at most one such relation between any pair of classes. The relation type together with the source and sink classes is thus sufficient to identify the relation uniquely. However, it is possible to have more than one association or aggregation relation between the same two classes, with different relations being distinguished by their names, which are essentially variable names except that the names `self` and `super` may not be used. Association and aggregation relations also have associated source and sink cardinalities, which may be one or many.

We use the type `Ref` to record the name and cardinalities of association and aggregation relations, the name being modelled using the type `Wf_Vble_Name` in order to exclude the unwanted values and the cardinality by the variant type `Card` which simply comprises the two possible values. The variant type `Relation_Type` similarly models the type of a relation: for inheritance and instantiation relations it is just a constant of the appropriate name, while for aggregation and association relations it is a function of the appropriate name applied to a value of type `Ref`. A relation as a whole is then modelled by the record type `Design_Relation`, which comprises the type of the relation and the names of its source and sink classes as explained above.

```
type
  Card == one | many,
  Ref ::
    relation_name : Wf_Vble_Name
    sink_card : Card
    source_card : Card,
  Relation_Type ==
    inheritance | association(as_ref : Ref) |
      aggregation(ag_ref : Ref) | instantiation,
  Design_Relation ::
    relation_type : Relation_Type
    source_class : Class_Name
    sink_class : Class_Name
```

The well-formedness condition `wf_relation` on a relation states that instantiation relations are not explicitly shown between a class and itself in the extended OMT diagram (every class is generally assumed to be able to instantiate itself) and that there cannot be inheritance relations between a class and itself since this would lead to essentially infinite inheritance structures. This is used as the basis for defining the subtype `Wf_Relation` of well-formed relations. The auxiliary function `is_assoc_or_aggr` simply checks whether a relation is either an association or an aggregation relation.

```
value
  wf_relation : Design_Relation → Bool
  wf_relation(r) ≡
    ~ is_assoc_or_aggr(r) ⇒ source_class(r) ≠ sink_class(r),

  is_assoc_or_aggr : Design_Relation → Bool
  is_assoc_or_aggr(r) ≡
    relation_type(r) ≠ inheritance ∧ relation_type(r) ≠ instantiation

type
  Wf_Relation = {| r : Design_Relation • wf_relation(r) |}
```

Because it is impossible to have two identical relations between the same two classes (we cannot have two instantiation relations or two inheritance relations between the same two classes, and if we have two association or aggregation relations between the same two classes then those two relations must have different names), it is sufficient to model the collection of all relations in a design using a set. However, this set of relations must satisfy certain constraints as follows:

`no_circularity`: it is not possible for a class to have a "meaningless" relationship with itself arising implicitly as a result of a loop of relations. For example, we cannot have a collection of relations in which classes c_1, ..., c_n are linked by inheritance relations in such a way that c_{i+1} is a subclass of c_i for all i and c_1 is a subclass of c_n;

`is_compatible_relation`: if two classes are related by an inheritance relation then there cannot be any other relation between the same two classes and in the same direction (though relations in the opposite direction are of course possible);

`different_variable_name`: association and aggregation relations must be uniquely identified by their names.

The function `is_valid_relation` combines the last two constraints together, and the function `is_correct_relation` then extends the properties from two relations to an arbitrary set of relations:

```
value
  is_valid_relation : Wf_Relation × Wf_Relation → Bool
  is_valid_relation(e1, e2) ≡
    is_compatible_relation(e1, e2) ∧ different_variable_name(e1, e2),

  is_correct_relation : Wf_Relation-set → Bool
  is_correct_relation(rs) ≡
    (∀ e1, e2 : Wf_Relation •
      e1 ∈ rs ∧ e2 ∈ rs ⇒ is_valid_relation(e1, e2))
```


Finally, we combine all constraints in the function `wf_relations` and use this as the defining predicate for the subtype `Wf_Relations` of well-formed sets of relations in the usual way:

```
value
  wf_relations : Wf_Relation-set → Bool
  wf_relations(rs) ≡ no_circularity(rs) ∧ is_correct_relation(rs)

type
  Wf_Relations = {| rs : Wf_Relation-set • wf_relations(rs) |}
```

11.2.3 A Formal Model of an Object-Oriented Design

A design as a whole then simply consists of a collection of classes and a collection of relations, the collection of classes being represented by the type `Classes` and the collection of relations by the type `Wf_Relations` defined in Section 11.2.2. A design is therefore modelled as a simple Cartesian product of these two types.

```
type
  Design_Structure = Classes × Wf_Relations
```

There are of course many constraints that must apply to this combination of classes and relations in order for it to correctly model an object-oriented design. These include, for example, that an abstract class must have subclasses, that state variables cannot be redefined in subclasses, that the sink of an instantiation relation cannot be an abstract class (because we cannot build instances of an abstract class), that a concrete class contains no abstract methods, etc. etc. All these constraints are combined into the function `is_wf_design_structure` and this forms the defining predicate of the subtype `Wf_Design_Structure` which then represents a general object-oriented design in the extended OMT notation:

```
type
  Wf_Design_Structure =
    {| ds : Design_Structure • is_wf_design_structure(ds) |}
```

11.3 Linking Designs to Patterns

A pattern represents an abstract "outline" or "skeleton" of a design. It shows a collection of classes, together with some of their properties and some relationships between the various classes.

In order for a part of a design to match a particular pattern, there must be classes in the design which have the same properties as each of those in the pattern and which are related in the same way. To be able to check whether a design matches a particular pattern we therefore need to link the model of a general object-oriented design described and specified above to the design patterns. In this section we restrict to defining how to link a design to a single instance of a single pattern. We extend this to matching a design against multiple instances of one or more patterns simultaneously in Section 11.5.

The relationship between a design and a pattern is defined using a *renaming map*, which associates the names of entities (classes, methods, state variables and parameters) in the design with the names of corresponding entities in the pattern. A typical example of this is shown in Figure 11.4. Note that because the pattern only represents an abstract skeleton of a design some entities in the design may have no corresponding entity in the pattern as is in fact the case in the example shown.

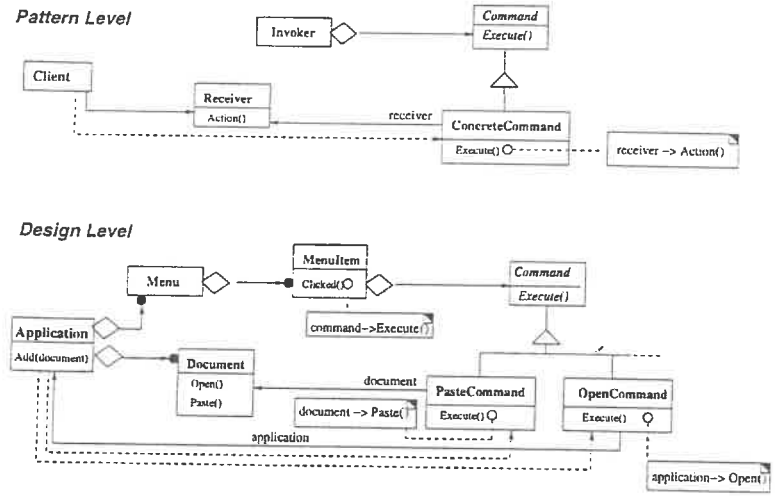


Fig. 11.4. Linking the design with the pattern.

The correspondence between state variables as well as that between parameters involves only a relationship between variables. We describe this using a *variable renaming*, which simply links names of variables in the design to those in the pattern. We model this using the simple map type `VariableRenaming`.

```
type
  VariableRenaming = Variable_Name ⇸ Variable_Name
```


The renaming for methods involves two parts. The first of these defines the correspondence between the names of the methods, while the second relates their parameters. Note that it is possible for several methods in the design to play the same *role* in the pattern, so we define the method renaming as the map type `Method_and_Parameter_Renaming`. This associates the names of methods in the design with instances of the type `Method_Renaming`, which consist of the method name in the pattern together with the variable renaming for the method's parameters. Note that the nested structure of the renaming is necessary because two different methods may have parameters with the same name.

```
type
  Method_Renaming ::
    method_name : Method_Name
    parameterRenaming : VariableRenaming,
  Method_and_Parameter_Renaming =
    Method_Name ⇸ Method_Renaming
```

The renaming of a class has a similarly nested structure, except that here we need to include both the map representing the renaming of the methods in the class and that representing the renaming of the state variables inside the record type `ClassRenaming` along with the name of the class in the pattern. The other difference is that in this case it is possible for a single class in the design to play several roles in the pattern (for instance, in the example illustrating the Command pattern in [55] the class Application in the design plays both the Client and the Receiver roles in the pattern). We therefore map each design class to a set of class renamings in the renaming map, and the full renaming map is represented by the type `Renaming`.

```
type
  ClassRenaming ::
    classname : Class_Name
    methodRenaming : Method_and_Parameter_Renaming
    varRenaming : VariableRenaming,
  Renaming = Class_Name ⇸ ClassRenaming-set
```

In order for the renaming map to be well-formed, no class in the design can have an empty set of renamings (we model the fact that a class in the design plays no role in the pattern by simply omitting it from the domain of the renaming map) and the renamings of any one design class must all refer to different pattern classes (otherwise a single design class can have two contradictory renamings).

We specify these two properties using the functions `images_not_empty` and `different_images_class_name` respectively, and these are combined in the function `is_wf_Renaming`, which then forms the defining predicate of the subtype `Wf_Renaming`.

```
value
  images_not_empty : Renaming → Bool
  images_not_empty(r) ≡ {} ∉ rng r,

  different_images_class_name : Renaming → Bool
  different_images_class_name(r) ≡
    (
      ∀ c : Class_Name, cr1, cr2 : ClassRenaming •
        c ∈ dom r ∧ cr1 ≠ cr2 ∧ cr1 ∈ r(c) ∧ cr2 ∈ r(c) ⇒
          classname(cr1) ≠ classname(cr2)
    ),

  is_wf_Renaming : Renaming → Bool
  is_wf_Renaming(r) ≡
    different_images_class_name(r) ∧ images_not_empty(r)

type
  Wf_Renaming = {| r : Renaming • is_wf_Renaming(r) |}
```

Then a design together with a renaming map which defines its correspondences to a given pattern is described by the simple Cartesian product type `Design_Renaming`.

```
type
  Design_Renaming = Wf_Design_Structure × Wf_Renaming
```

This combination must satisfy the following constraints:

`is_correct_domain`: every entity which has a renaming under the renaming map must be in the design;

`equal_meth_ren_in_hierarchy`: if a method plays some role in a superclass and also some role in a subclass then the two roles must be the same;

`meth_ren_in_hierarchy`: if a method plays some role in a superclass and the subclass inherits a different version of the method (i.e. the method is redefined locally or in an intermediate class) then the method must also play a role in the subclass;

`card_images_of_parameters`: no method has more than one parameter. (This constraint applies only to matching designs against GoF patterns and derives from the fact that no method in the GoF patterns has more than one parameter. It would be omitted if we wanted to generalise the matching to other forms of patterns.)

The function `is_wf_design_renaming` combines these constraints together, and this is then used to define the subtype `Wf_Design_Renaming` of `Design_Renaming` in the usual way.


```
value
  is_wf_design_renaming : Design_Renaming → Bool
  is_wf_design_renaming(dr) ≡
    card_images_of_parameters(dr) ∧
    is_correct_domain(dr) ∧
    equal_meth_ren_in_hierarchy(dr) ∧ meth_ren_in_hierarchy(dr)

type
  Wf_Design_Renaming =
    {| pr : Design_Renaming • is_wf_design_renaming(pr) |}
```

11.4 Specifying Properties of Patterns

The description of each particular pattern in [55] defines the properties of the various components (classes, methods, and relations) in that pattern. We can formally specify these properties for each individual pattern using our model. We illustrate this process for the Command pattern, the structure of which is shown in Figure 11.1. Corresponding specifications of most of the other GoF patterns can be found in [100, 51, 118, 50, 6].

The structure essentially 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.

The interaction diagram in Figure 11.2 shows that the Client 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 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 pattern structure (Figure 11.1).

It is in fact quite common for information about different aspects of a pattern to be given in different parts of its description in [55]. However, this can make it difficult for a designer to be sure that a given design captures all aspects of the pattern correctly. We therefore base our formal specification on an extension of the pattern structure which is shown in Figure 11.5.

This includes the full invocation as shown in the interaction diagram in Figure 11.2, and also introduces a new role, ClientMethod, into the Client class to represent the method which invokes the StoreCommand method in the Invoker, which we also show explicitly. However, we allow this invocation

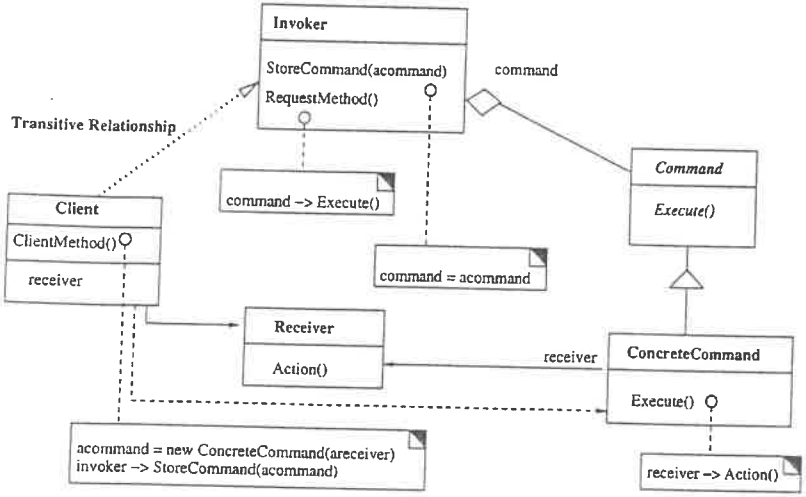


Fig. 11.5. The modified Command structure.

to be indirect, so that the relation between the Client and the Invoker may be transitive. We also explicitly define the body of the StoreCommand method based on its properties shown in the interaction diagram – it should have a single parameter which represents a command and it simply assigns this parameter to its command state variable.

Note that our modified pattern structure omits the state variable state from the ConcreteCommand class. This is because, according to the discussion of the Command pattern in [55], this variable is intended to support undo and redo operations and the pattern can in fact be used to design systems which do not have such operations[3].

The first step towards formally specifying the properties of the pattern is to introduce constants which represent the names of the classes, methods and variables appearing in the pattern and to use axioms to ensure that these values are all different[4]. These constants are:

```
value
  Invoker : Class_Name,
  Client : Class_Name,
```

[3] The state variable is in any case not sufficient to support undo and redo operations on its own. Additional methods would also be required in the Command and/or ConcreteCommand classes.

[4] These axioms are omitted here for brevity. They are entirely analogous to the similar axiom introduced in Section 11.2.2 to ensure that the two reserved variable names `self` and `super` are distinct.


```
  Command : Class_Name,
  ConcreteCommand : Class_Name,
  Receiver : Class_Name,
  command : Variable_Name,
  receiver : Variable_Name,
  command_param : Variable_Name,
  Execute : Method_Name,
  Action : Method_Name,
  StoreCommand : Method_Name,
  ClientMethod : Method_Name,
  RequestMethod : Method_Name
```

Next we specify the properties of the various entities in the patterns.

Rather than specifying the properties of each pattern directly, however, we try to identify properties which are common to several patterns and specify these more generally (i.e. without explicit reference to any one pattern). This helps to make our specification not only shorter but also more general and easier to extend to new patterns – the generic functions we introduce encapsulate common properties in such a way that they only need to be specified once and, being generic, are more likely to be reusable if we later wish to extend our specification to new patterns.

Looking at the Command and ConcreteCommand classes in the Command pattern, we can see that these are related by an inheritance relation, the Command class being an abstract superclass of the ConcreteCommand class. Many patterns contain a pair of classes which have precisely these properties, so we introduce a generic function `hierarchy` which captures them and which has the names of the roles involved as parameters. Recall, however, that the pattern represents an idealised design and a concrete design may include other classes. In this particular instance this means first that there may be more than one subclass playing the ConcreteCommand role and second that there may be intermediate classes in the hierarchy (i.e. that the ConcreteCommand classes are not necessarily direct subclasses of the Command class. We incorporate this freedom into our specification of the `hierarchy` function, and the function thus 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 (Command), has leaf classes which play any of a given set of roles in the pattern (ConcreteCommand), does not contain classes playing other roles in the pattern, and has roles which are consistent in superclasses and subclasses (i.e. if some subclass of the root class plays a particular role in the pattern and some subclass of that subclass also plays a role in the pattern then the two roles must be the same). The actual specification of the function `hierarchy` is quite long so we show only its signature here. The required property of the Command pattern is then embodied in the function `Command_hierarchy` which simply instantiates the function `hierarchy` with the appropriate roles.

```
value
  hierarchy : Class_Name × Class_Name-set ×
    Class_Name-set × Wf_Design_Renaming → Bool
  hierarchy(cp1, cps1, cps2, ((dsc, dsr), r)) ≡ ... ,

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

Other properties of the classes, methods and relations in the Command pattern are specified in a similar way. Thus, for example, every class playing the ConcreteCommand role should contain exactly one state variable playing the receiver role, which is specified generically by the function `store_unique_vble` and for the Command pattern by the function `store_receiver` which is simply defined as the appropriate instantiation of this:

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

Similarly, every class playing the Invoker role should contain at least one method playing the RequestMethod role, and every method playing this role in the class should be implemented and should contain an invocation to the command state variable of the method which plays the Execute role in the Command class. Furthermore, every class playing the Invoker role should have exactly one state variable which plays the command role and should be linked to the class playing the Command role by an aggregation relation representing the state variable playing the command role and there should be no other relations between classes of these two roles. For the properties of the methods playing the RequestMethod role, the function `exists_method` checks generally whether a class has some method playing a given role, while the function `deleg_with_var` checks that every method playing a particular role in a particular class is implemented and contains an invocation to a particular state variable in the same class of a given method in some other given class. For the properties of the state variable and the relations, the function `store_unique_vble` states that a class should have only one state variable playing a given role, the function `has_unique_assoc_aggr` checks that there

is precisely one association or aggregation relation and no instantiation relations between two classes, and the function `has_assoc_aggr_var_ren` checks that the relation has given type and arity and corresponds to a state variable playing a particular role. These functions, appropriately instantiated, are combined in the functions `Invoker_invoke` and `Invoker_has_Command` which represent the required properties of the Command pattern.

```
value
  Invoker_invoke : Wf_Design_Renaming → Bool
  Invoker_invoke(ds, r) ≡
    exists_method(Invoker, RequestMethod, r) ∧
    deleg_with_var
      (Invoker, RequestMethod, command, Command, Execute, (ds, r)),

  Invoker_has_Command : Wf_Design_Renaming → Bool
  Invoker_has_Command(dr) ≡
    has_unique_assoc_aggr(Invoker, Command, dr) ∧
    has_assoc_aggr_var_ren
      (Invoker, Command, Aggregation, command, one, dr) ∧
    store_unique_vble(Invoker, command, dr)
```

The other properties of the Command pattern are specified analogously, and then combined together into the function `is_command_pattern` which then gives the complete specification of the properties of the pattern.

```
value
  is_command_pattern : Wf_Design_Renaming → Bool
  is_command_pattern(dr) ≡
    Command_hierarchy(dr) ∧
    store_receiver(dr) ∧
    Invoker_has_Command(dr) ∧
    Invoker_invoke(dr) ∧
    ...
```

Specifications like the above can be used to formally define whether a subset of a given design matches a given pattern – the renaming map introduced in Section 11.3 defines the correspondence between entities in the design and entities in the pattern (or which role in the pattern is played by a particular entity in the design), so the design matches the pattern if each entity in the renaming map at the design level satisfies the properties of the pattern level entity to which it renames. A concrete example of this process can be found in [52].

11.5 Extending to Multiple Patterns

The model of the renaming map presented in Section 11.3 above assumes that only a single instance of a single pattern is being linked to the design at any time. In reality, of course, a given design may involve many different patterns and also many different occurrences of the same pattern. In addition, the patterns may be overlapping, with a single class or method in a design playing one role in one pattern and another role in another pattern. In this section we extend the renaming map so that it can link a single design with a group of patterns simultaneously.

In order to do this, however, we need to know not just the role that a particular class plays but also the pattern in which it plays that role – this is because different GoF patterns contain the same role so in general it is impossible to deduce the pattern from the name of the role alone. In addition, we need to take account of the facts that one class can play different roles in different patterns and that many instances of the same pattern may occur in a single design.

We model each instance of each particular pattern using the same renaming map as we used in the matching of a design to a single pattern (the type `Wf_Renaming` defined in Section 11.3) and we introduce the abstract type `RenId` of *renaming identifiers* to distinguish the renamings associated with different occurrences of the same pattern in the design. The collection of renamings associated with one given pattern is then modelled using the map type `Renaming_Map` which associates renaming identifiers with well-formed renamings. Then the *multi-renaming*, which is the renaming for all instances of all patterns in the whole design, is represented as a map from *pattern names* to the collection of renamings corresponding to that pattern, where the pattern names are represented by the sort type `Pattern_Name`. This basically comprises the set of names of all the GoF patterns, but it is specified more abstractly to allow more patterns to be added to the specification if necessary. Thus we specify simply that the names `Abstract_Factory`, `Adapter_Class`, `Adapter_Object`, `Bridge`, and so on are all valid but different pattern names but we do not specify that these are the only such names[5].

```
type
  Pattern_Name,
  RenId,
  Renaming_Map = RenId ~m~> Wf_Renaming,
  Multi_Renaming = Pattern_Name ~m~> Renaming_Map

value
  Abstract_Factory, Adapter_Class, Adapter_Object, Bridge, Builder,
```

[5] Again we omit the axioms for brevity since they are once more entirely analogous to the similar axiom introduced in Section 11.2.2 to ensure that the two reserved variable names `self` and `super` are distinct.


```
  Command, Composite, Decorator, Facade, Factory_Method, Flyweight,
  Iterator, Mediator, Memento, Observer, Prototype, Proxy,
  Singleton, State, Strategy, Template_Method, Visitor
    : Pattern_Name
```

Next, we associate each pattern name with the formal specification of that pattern, in particular with the functions like `is_command_pattern` which specify the complete set of properties of the patterns, in this case the Command pattern. This is done using the *pattern table* which simply maps the pattern name to the appropriate function.

```
type
  Pattern_Table = Pattern_Name ~m~> (Wf_Design_Renaming → Bool)

value
  pattern_table : Pattern_Table =
    [ Command ↦ is_command_pattern,
      ...
    ]
```

If a particular pattern does not occur in a given design, we can indicate this by simply omitting the pattern name from the domain of the multi-renaming. We therefore rule out the possibility that a pattern has an empty collection of renamings in the multi-renaming, which would basically also amount to the pattern not occurring in the design, by imposing an appropriate well-formedness constraint. In addition, we require that only "known" patterns, that is patterns whose names occur in the pattern table, may be included in the multi-renaming. This is important because in order to check whether a part of the design matches a pattern we need to know which function defines the properties of that pattern.

These two constraints are combined in the well-formedness function `is_wf_multi_renaming` and this function is then used to construct the subtype `Wf_Multi_Renaming` of well-formed multi-renamings.

```
value
  is_wf_multi_renaming : Multi_Renaming → Bool
  is_wf_multi_renaming(mr) ≡
    [] ∉ rng mr ∧ dom mr ⊆ dom pattern_table

type
  Wf_Multi_Renaming =
    {| p : Multi_Renaming • is_wf_multi_renaming(p) |}
```

A well-formed design together with a multi-renaming then forms the basis for checking whether a design matches a collection of patterns simultaneously. This is represented by the type `Multi_Design_Pattern` (cf. the definition of the type `Design_Renaming` in Section 11.3; we have simply replaced

the second component Wf_Renaming of the Cartesian product with the type Wf_Multi_Renaming).

```
type
  Multi_Design_Pattern = Wf_Design_Structure × Wf_Multi_Renaming
```

In the simple version of the model described in Section 11.3, we required that the renaming should be consistent with the design to which it is being applied (the function is_wf_design_renaming). In moving from a single renaming to a multi-renaming we have essentially replaced the single renaming map with a collection of renaming maps, so we similarly require that each of the renaming maps in this collection is consistent with the design, that is that the relationship between the design level and the pattern level that the multi-renaming defines is consistent. This means that each of the renaming maps in the multi-renaming must individually satisfy the well-formedness condition is_wf_design_renaming with respect to the design. This property is captured in the function each_renaming_is_wf:

```
value
  each_renaming_is_wf : Multi_Design_Pattern → Bool
  each_renaming_is_wf(ds, mr) ≡
    (
      ∀ p : Pattern_Name, r : Wf_Renaming •
        p ∈ dom mr ∧ r ∈ rng mr(p) ⇒ is_wf_design_renaming(ds, r)
    )
```

We also require that each of the renamings associated with a single pattern in the multi-renaming must represent a different instance of the pattern, that is there cannot be two renamings for the same pattern in which every class plays the same role in both renamings. This property is embodied in the function each_instance_is_different, though its specification is made in terms of several auxiliary functions so is omitted here for brevity. The well-formedness constraint on the type Multi_Design_Pattern is then simply the conjunction of the two functions each_renaming_is_wf and each_instance_is_different. The function is_wf_multi_design_pattern combines these constraints and this is then used to define the subtype Wf_Multi_Design_Pattern of Multi_Design_Pattern:

```
type
  Wf_Multi_Design_Pattern =
    {| p : Multi_Design_Pattern • is_wf_multi_design_pattern(p) |}

value
  is_wf_multi_design_pattern : Multi_Design_Pattern → Bool
  is_wf_multi_design_pattern(mdp) ≡
    each_renaming_is_wf(mdp) ∧ each_instance_is_different(mdp)
```


With this extension, it is now possible to check whether a subset of a design matches a collection of patterns simultaneously. This is analogous to the method explained for a single pattern at the end of Section 11.4 except that in this case we require that for each renaming in the multi-renaming the design-level entities satisfy all the properties of the pattern-level entities to which they rename.

11.6 Conclusions

In this chapter we have described a formal model of a generic object-oriented design based on the extended OMT notation and we have shown how a design in this model can be linked to a GoF pattern using the renaming map. We have furthermore shown how the specific properties of individual GoF patterns can be specified in this model, and we have briefly indicated how such specifications can be used to determine whether or not a given design matches a given pattern. Finally, we have shown how it is possible with only slight modifications to extend the formal model from one which matches only a single design pattern against a subset of a design to one which matches a design against a set of patterns, possibly with overlapping roles and also allowing multiple instances of the same pattern.

The model allows designers to be sure, as well as to demonstrate to others, that they are using the patterns correctly and consistently. It can also help designers to understand the properties of the GoF patterns more clearly – indeed developing our specifications of the patterns such as that of the Command pattern discussed in Section 11.4 has identified a number of inconsistencies and incompletenesses in the informal descriptions of a number of patterns and has led us to propose similar modified or extended pattern structures for other patterns which resolve these problems.

Although we have limited our attention to GoF patterns in this work, we believe that the basic model of object-oriented design which we have presented in Section 11.2.2 is sufficiently general that it could be applied in a similar way to give formal descriptions of other design patterns based on the extended OMT notation. We also believe that the model could be modified relatively easily to give a similar model of object-oriented designs based on the UML notation (http://www.omg.org/uml).

Finally, we believe that the formality of our general model and of our specifications of the individual GoF patterns makes them a useful basis for tool support for GoF patterns and this is currently under investigation.

Background Information

This chapter is based on work undertaken by four members of the Facultad de Economía y Administración, Universidad Nacional de Comahue, Neuquén,

Argentina – Alejandra Cechich, Andres Flores, Luis Reynoso, and Gabriela Aranda – in collaboration with Richard Moore.

The first phase was carried out with Alejandra Cechich [21, 20] and concentrated on formally specifying the properties of some of the GoF patterns directly in RSL, concentrating particularly on the responsibilities and collaborations of the pattern participants. Here the structural aspects of the patterns were specified statically while the collaborations were specified in terms of sequences of interactions, thus closely mirroring the information presented in the interaction diagrams.

The generalisation of the model to a general object-oriented design and how to link a design with a single pattern using the renaming map was done with Andres Flores and Luis Reynoso [52, 53]. This model, which is the one presented here, represents both the structural properties and the collaborations statically, the collaborations being defined partly by the relations between the classes and partly by the requests the methods make to other classes through their bodies.

The specifications of the properties of the individual GoF patterns in the general model was done with Luis Reynoso (behavioural patterns; [100, 118]), Andres Flores (structural patterns; [51, 50]), and Gabriela Aranda (creational patterns; [6]).

Finally, the extension of the model to match a design against many patterns simultaneously, as well as work on specifying the properties of particular examples of so-called "compound patterns" (see for example [120, 141, 142, 143]) which is not presented here, was done with Gabriela Aranda [5].