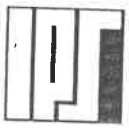

Proceedings of the ACIS
2nd International Conference on

Software Engineering, Artificial Intelligence, Networking & Parallel/Distributed Computing

SNPD '01

August 20-22, 2001
Nagoya Institute of Technology, Japan

Edited by:

Naohiro Ishii, Nagoya Institute of Technology, Japan
Tadanori Mizuno, Shizuoka University, Japan
Roger Y. Lee, Central Michigan University, U.S.A.

ISBN:0-9700776-1-0

A Publication of the International Association for Computer and Information Science

Sponsor: The International Association for Computer and Information Science
Office:735 Meadowbrook, Mt. Pleasant, MI 48858, U.S.A.
URL:http://acis.lsfk.org E-mail:acis@lsfk.org Phone:(517)744-3811

A Precise Specification of GoF Behavioural Patterns

Richard Moore
United Nations University
International Institute for Software Technology
P.O. Box 3058, Macau
tel: +853-712930
email: rm@iist.unu.edu

Luis Reynoso
Universidad Nacional del Comahue
Dep. de Informatica y Estadistica
Buenos Aires 1400, 8300 Neuquen. Argentina
tel: +54-299-4490312
email: lreynoso@uncoma.edu.ar

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. Eleven such patterns, which can be used to capture different aspects of behaviour in a design, are described in the GoF catalogue. However, this description is essentially informal, consisting of a combination of a graphical notation based on an extension of OMT together with natural language and sample code, and many of the essential properties of the pattern constituents are implicit, for example in the names of the classes and methods in the pattern. It is therefore not always easy for a designer to be sure that a particular design exhibits all the essential properties of a pattern. In this paper, we describe a detailed analysis of the GoF behavioural patterns which attempts to identify all their essential properties explicitly, and we show how these can also be formally specified using a formal model of generic object-oriented designs. We also discuss various issues relating to the implicit properties of the patterns that arose during this analysis.

1. Introduction

Design patterns are the product of one cognitive intellectual activity, abstraction, "a fundamental objective of good software development" [5]. The patterns are generic and embody "best practice" solutions to a particular range of design problems, and although these solutions are not necessarily the simplest or most efficient for any given problem they are nevertheless proven solutions to particular aspects of design. The patterns thus offer designers a way of reusing these existing solutions rather than having to start each new design from scratch. They also provide designers with an effective "shorthand" for communicating with each other about complex concepts: 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 particular application domain, are the so-called GoF[1] patterns, described in [8]. 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, 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.

The patterns in the GoF catalogue are described using a consistent format which has in fact effectively been adopted as the standard way of presenting software design patterns. This uses a graphical notation based on an extension of OMT (Object Modelling Technique [13]) to represent the main constituents of the pattern – classes, methods, and relationships between classes – and supplements this 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 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 in-

[1] Gang of Four.

tentionally 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 more precise specification of the properties of the patterns can not only help to alleviate these problems but can also improve understanding of the patterns in general, particularly if a more precise notation is also used to describe the properties. One approach to this [3] represents patterns as formulae in LePUS, a language defined as a fragment of higher order monadic logic [4]. A second [10] formalises the temporal behaviour of patterns using the DisCo specification method, which is based on the Temporal Logic of Actions [9]. A third [2, 1] uses the RAISE Specification Language (RSL; [11]) to formally specify properties of the patterns, in particular the responsibilities and collaborations of the pattern participants.

Our approach (see [7] for full details) is based on that of [2, 1], though we have significantly extended the scope of the model used therein. First, we have decoupled the design level from the pattern level by developing an abstract but formal model of a general object-oriented design in terms of classes, their properties, and the relationships between them. A design is then "matched" against a pattern by using a *renaming map* to associate the various elements of the design (classes, state variables and methods, including their input parameters and results) with the names of the entities appearing in the pattern. 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. Second, our model supports the specification of the behavioural properties of the design, specifically the actions that are to be performed by the methods, which could not be specified in the model used in [2, 1]. Finally, we comprehensively analyse and specify the essential components and properties of the individual 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. We make these properties explicit by formulating extensions to the presentation of the structure of the patterns.

In this paper, we describe the results of our analysis of the GoF behavioural patterns, which characterize the way in which classes and objects interact and distribute responsibilities [8]. We begin by illustrating our approach using one of the simpler behavioural patterns, the Template Method pattern, as an example. We explain our analysis and specification of this pattern in some detail in Section 2. Then in Section 3 we discuss in more general terms other issues that arose in our analysis of the other behavioural patterns, particularly those relating to the implicit properties of the patterns. We also discuss the ways in which we dealt with these issues in our analysis and specification. We conclude with a brief summary of our work and an indication of how we plan to extend it in the future.

2. Analysis of the Template Method Pattern

In [8], the main constituents (classes, methods, relationships, etc.) of a pattern are shown in its *structure*, which represents these entities using an extended version of OMT [13]. The structure of the Template Method pattern is shown in Figure 1.

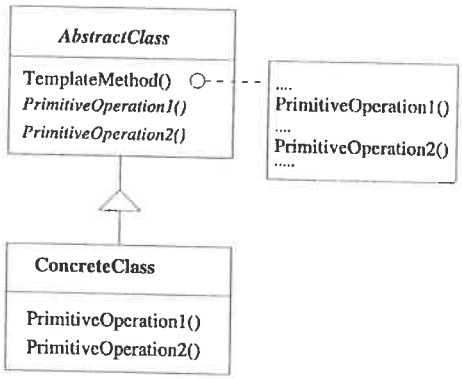

Figure 1. Template Method Pattern Structure

This is supplemented by textual descriptions of the purpose of the pattern (its *intent*), the functions performed by each of the classes and/or objects in the pattern (the *participants*), and the essential communications between these participants (the *collaborations*). For the Template Method pattern these are 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.

However, the pattern structure shown in Figure 1 is idealised and in fact a design with a more general structure would serve equally well as an implementation of the pattern. For example, the structure shows the class called ConcreteClass as an immediate subclass of the class called AbstractClass whereas there could be other classes intermediate between them in the design, these intermediate classes effectively having no direct counterparts in the pattern and thus playing no specific roles in the pattern. Also only one subclass of the class AbstractClass is shown whereas in practice a design is almost certain to include many such subclasses, otherwise the abstract superclass does not serve a very useful purpose (with only one subclass, the subclass and the superclass could be combined into a single class without losing anything).

Thus, we generalise the description of the pattern to allow not only intermediate classes but also multiple classes which play the ConcreteClass role. To this effect, we consider the structure of the Template Method pattern to consist of an *inheritance hierarchy*, the root of which corresponds to the AbstractClass role and the leaves of which play the ConcreteClass role. This structure corresponds more closely to the intent of the pattern – the abstract class implements a method in terms of operations which are defined locally but implemented in subclasses.

We also permit intermediate classes in the hierarchy to play the ConcreteClass role – this covers the possibility that different leaf classes in the hierarchy in the design actually have the same implementation of the PrimitiveOperation methods, in which case these methods might well be implemented in a common superclass.

The pattern structure in Figure 1 also shows only a single class playing the AbstractClass role and a single method belonging to that class which plays the TemplateMethod role. We could of course have more than one class in a design which plays the AbstractClass role, and each such class could also include more than one method which plays the TemplateMethod role: one design could of course use the Template Method pattern many times. However, without loss of generality we can consider each of these uses of the pattern as an independent and individual instance of a "basic" Template Method pattern in which only one class plays the AbstractClass role and that class contains only one method which plays the TemplateMethod role. We therefore make these restrictions in our model. Of course a single TemplateMethod can depend on more than one PrimitiveOperation and these can be implemented at different levels within the hierarchy.

The properties of the other entities in the Template Method pattern are to a large extent as shown in the structure in Figure 1. We thus arrive at the following precise, yet informal statement of the pattern's properties:

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.

In order to make the above properties even more precise, we additionally give a formal specification of them based on our formal model of object-oriented designs and patterns [7], which is written using the RAISE Specification Language, RSL [11]. There are basically two levels to this model: the *design level*, which defines the basic constituents of a design (classes, methods, relations, instance variables, etc.) as well as consistency properties that they must satisfy, and the *pattern level*, which defines common properties of the classes and relationships appearing in the various patterns in the GoF catalogue. These common properties are appropriately parameterized over the names of the entities involved, and the properties of a particular pattern are then expressed as combinations of these properties, the names being suitably instantiated with the relevant names from the pattern.

Thus, for example, two of the common properties defined at the pattern level in the model are expressed by 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 of it and none of the methods that are visible in the classes (including inherited methods) are abstract. Note that this second function does not require that the subclasses must be immediate subclasses: they could in fact be separated from the parent class by an arbitrary number of intermediate classes.

```
exists_role :
    Class_Name × Wf_Design_Renaming → Bool
exists_role(cp, ((dsc, dsr), r)) ≡
    (
        ∃ cd : Class_Name •
            renaming_class_name(cd, cp, r)
    ),

is_concrete :
    Class_Name × Class_Name × Wf_Design_Renaming
        → Bool
is_concrete(cp1, cp2, ((dsc, dsr), r)) ≡
    (
        ∀ cd1, cd2 : Class_Name •
            renaming_class_name(cd1, cp1, r) ∧
            renaming_class_name(cd2, cp2, r) ⇒
            is_superclass(cd1, cd2, dsr) ∧
            is_concrete_class(cd2, (dsc, dsr))
    )
```

Instantiating these generic functions appropriately, in the first case with the role ConcreteClass and in the second case with the roles AbstractClass and ConcreteClass, then yields a formal specification of the second property of the Template Method pattern listed above:

```
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)
```

Similarly, the first property of the Template Method pattern is expressed by appropriately instantiating the generic function *hierarchy* 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, and has no classes which play roles from some other given set of roles. In the Template Method pattern the root of the hierarchy must play the AbstractClass role, the leaves the ConcreteClass role, and no roles are excluded.

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

The other properties of the Template Method pattern are specified similarly.

The design level and the pattern level are linked by the renaming map, which associates various elements of a design (classes, state variables and methods, including their input parameters and results) with the names of corresponding entities in the pattern. 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. This allows us to 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. In this way, the pattern level can be thought of as a meta-model of design – our specifications define the properties that a design must exhibit if it is to be considered to match a particular pattern, and in general a wide range of designs will satisfy these properties. A detailed example illustrating this matching process can be found in [12].

We have so far analysed and specified the properties of nine of the eleven GoF behavioural patterns in the same way (see [12] for full details). This analysis revealed several areas where the properties of the patterns are not dealt with clearly or explicitly in [8] but are rather implicit, for example in the intent and motivation of the pattern or in the names used for the entities in the pattern. We discuss some of these issues in the following section.

3. Specifying Implicit Properties of Behavioural Patterns.

In the discussion of the Template Method pattern presented in the previous section, we have already seen how our analysis can identify properties of the patterns which are not only not stated explicitly in the GoF catalogue [8] but which in some cases cannot even be represented in the extended OMT notation which is the standard way of presenting the patterns – we allow an arbitrary number of intermediate classes to occur between the class playing the AbstractClass role and that playing the ConcreteClass role; we allow methods to be inherited from superclasses rather than insisting that they should be defined locally; and we discover that without loss of generality we can assume that there is only one class playing the AbstractClass role and only one method playing the TemplateMethod role. In this section, we discuss similar issues relating to generalisations and resolution of ambiguities which arose from our analysis of the other behavioural patterns. We refer the reader to [8] for the detailed descriptions of these patterns, including their structures and the names of the entities which appear in them which we quote here generally with little or no explanation. We also refer the reader to [12] for full details of the analysis and specifications of the patterns.

3.1. Specifying Cardinalities of Roles

We have already seen in the analysis of the Template Method pattern how we specify the number of entities in a design which may play a particular role in a pattern, i.e. the *cardinality* of each role. Thus, for example, we specified that for each "instantiation" of the Template_Method pattern there should be only one class in the design which plays the AbstractClass role and only one method which plays the TemplateMethod role but that there could be more than one class playing the ConcreteClass role and more than one method playing the PrimitiveOperation role.

In the OMT diagram of the structure shown in Figure 1, the fact that there can be more than one method playing the PrimitiveOperation role is indicated by the explicit inclusion of the two method names "PrimitiveOperation1" and "PrimitiveOperation2". However, in many patterns the cardinality of the roles is not explicitly indicated in this way and one has to analyse the description of the pattern in some detail in order to determine this.

An example of this comes in the State and Strategy patterns, which have almost identical structures in [8]: 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 up to renaming of their constituents. This similarity can be captured explicitly and very concisely in the formal model by simply applying a formal renaming to the specification of the State 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
```

However, the intent of the Strategy pattern is completely different from that of the State pattern, and we can identify one important difference between the two patterns which is not apparent from the structure but which derives from their different intents. 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 [14]. 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. The AlgorithmInterface method in the Strategy pattern thus has cardinality one whereas the Handle method in the State pattern has cardinality many.

Similar considerations apply to the cardinalities of state variables and parameters. Thus, for example, in the Command pattern each class playing the ConcreteCommand role may contain many state variables which play the state role but each concrete command has a unique receiver so it contains precisely one state variable which plays the receiver role. Similarly, in the Memento pattern both the GetState and the SetState methods in the Memento class are shown without parameters in the structure in [8]. This is correct for GetState, which indeed requires no parameters, but SetState clearly requires parameters and in fact the number of parameters it requires is equal to the number of state variables playing the state role in the Memento class (see Section 3.3 below for further discussion of this point).

3.2. Degeneracy and Variants of Patterns

In the Mediator pattern, the structure in [8] shows an abstract class Mediator which has a concrete subclass ConcreteMediator. However, a design in which there is a single class which plays the ConcreteMediator role conforms to this structure and hence represents a valid instantiation of the pattern, although in practice there is little to be gained by separating the Mediator and ConcreteMediator roles in the design in this case and we could combine them into a single role, which would of course be the ConcreteMediator role, and omit the Mediator role entirely – indeed in the discussion of the implementation of the Mediator pattern in [8], it is stated explicitly that "there is no need to define an abstract Mediator class when colleagues work with only one mediator".

The situation is of course the same in all the patterns which contain an inheritance hierarchy. We choose not to rule out these degenerate situations, however, on the grounds that it is certainly not incorrect to include a class in a design which has only one subclass. We therefore still consider this case to be a valid instantiation of the pattern, and we regard the case in which the abstract class is omit-

ted from the pattern as a *variant* of the pattern which is to be specified independently. As a result, our specification of the relevant property in the Mediator pattern requires simply that there should be at least one class in the design which plays the ConcreteMediator role:

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

In fact, similar variations can be made to many patterns, and indeed the concept of variants of a pattern is used by many authors when they want to express a refinement or an extension of a pattern. Analysis and specification of these variants will be subject of future work.

3.3. Modelling Alternative Implementations

In the structure of the Memento pattern presented in [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: the Originator effectively sees and treats the Memento as a single entity and does not need to know about or manipulate its internal structure. Indeed in the description of the Caretaker role in the participants of the Memento pattern in [8] it is explicitly stated that "Caretaker never operates on or examines the contents of a memento". 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 in the structure (see Figure 2).

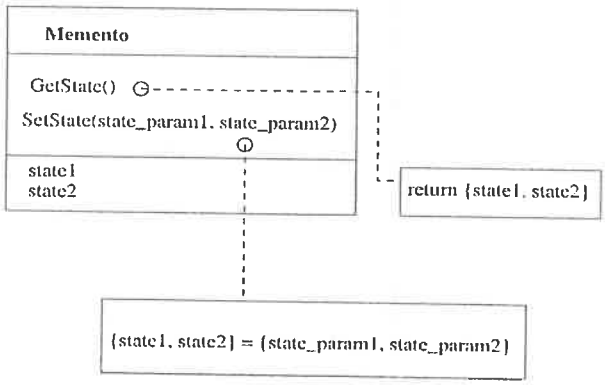

Figure 2. The Extended Memento Class

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. However, in this case the reason for the choice is comparitively weak and the alternative implementation using a parameterized new method, perhaps also with the SetState method omitted from the pattern, would in fact be an equally good design and could be considered as a variant of the pattern.

A similar situation arises in the Observer pattern, where the SetState and GetState methods in the ConcreteSubject class perform similar tasks to the methods with the same names in the Memento class. 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, pro-

vided of course that each subjectState variable is set and returned by at least one of each of those methods. In this

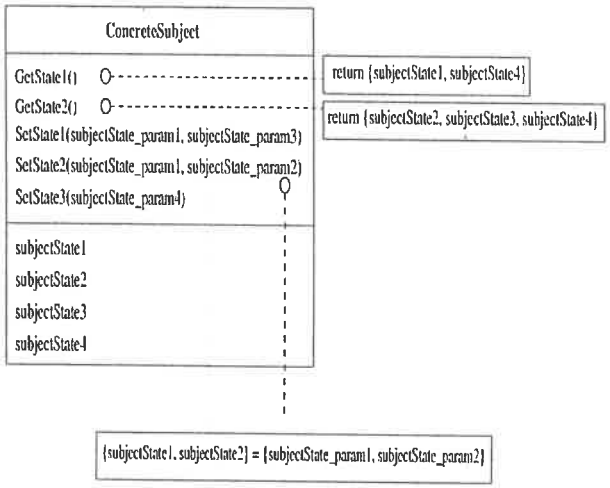

Figure 3. The Extended ConcreteSubject Class

case we model the methods as generally as possible, in fact giving an abstract specification which covers all possible implementations. We therefore specify only 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 specify 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 specify 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 details are again included in the pattern structure by adding appropriate annotations to the methods (see Figure 3).

3.4. Comparing Quantities of Entities

As mentioned in the previous section in the discussion of the GetState and SetState methods in the Memento pattern, the fact that we specify that there should be only one of each of these methods means that the number of parameters of the SetState method should be the same as the number of state variables in the Memento class (because the method must define the whole of the state of the Memento class at once). We therefore include this property explicitly in our specification.

In the above case, we are specifying a relationship between the number of parameters of a method and the number of state variables in a class. In other patterns there are similar relationships between the number of methods and the number of classes (Visitor pattern), between the numbers of state variables in different classes (Memento pattern again), and between the number of methods and the number of relations (Iterator pattern).

In the Visitor pattern 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 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.

In the Memento pattern, the Originator class creates a memento containing a snapshot of its current internal state, and can later use that memento to restore its internal state. The class playing the Originator role and the class playing the Memento role must therefore have the same number of state variables which play the state role.

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.

Again, we explicitly include these properties in our specifications of these patterns.

3.5. Modelling Indirect Invocations

The client has an important role in the Command pattern as shown in the interaction diagram in Figure 4 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 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 in [8].

Part of this interaction is shown in the OMT diagram representing the application of the Command pattern dis-

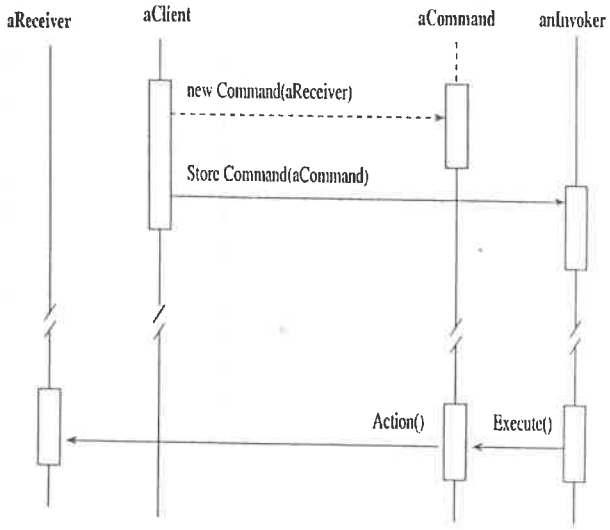

Figure 4. Collaborations of the Command Pattern

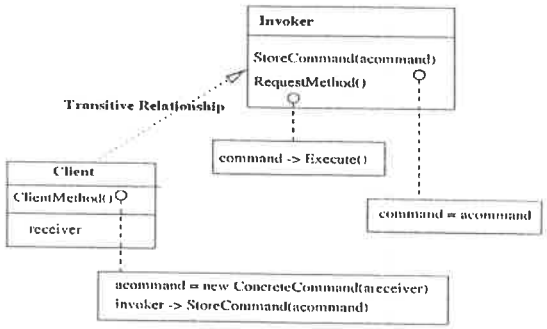

Figure 5. The Interaction between Client and Invoker

cussed in the motivation of the pattern in [8], 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 4. We first introduce a new state variable command into the Invoker class which is responsible for storing the command which is invoked by the Invoker, and we also introduce the method StoreCommand which is responsible for initialising this variable. The StoreCommand method therefore has a single command as its parameter and its body simply assigns this parameter to the command state variable. Then we require that the Client class contains a method which invokes this StoreCommand method in the Invoker, though we allow this invocation to be indirect. We introduce another new role, ClientMethod, this time into the Client class, to represent this method and we also introduce a (possibly) transitive relationship between the Client and the Invoker classes. These additions are again represented in a modified OMT diagram representing the pattern structure as shown in Figure 5.

4. Conclusions

We have shown how a detailed analysis of the properties of the GoF behavioural patterns can reveal new information about the patterns, including information that is implicit in the description, intent, motivation, class and method names and so on of the pattern. We have also shown how the properties of the patterns can be formally specified, using an extension of the generic formal model of object-oriented design described in [7].

Our analysis results in a concise description of the properties of the patterns in the form of a list of the essential characteristics of each of the entities in the pattern, which can serve as a check list for designers who wish to check that they are applying the patterns correctly. In many cases we have also formulated extensions to the structure of the patterns as presented in [8], though this is not always possible because some of the additional properties we specify cannot be represented in the extended OMT notation. This presentation thus preserves the descriptions of the patterns in terms of a combination of graphical notation and natural language which many designers prefer to use.

Our analysis and specification so far covers nine of the eleven GoF behavioural patterns [12], while related work addresses the GoF structural patterns [6] and creational patterns. In future work we intend to consider variants of the patterns and also to extend the model slightly so as to allow different subsets of a design to be compared against several patterns simultaneously. In addition, since the formal specifications of the patterns provide a concise and unambiguous description of their essential properties, we plan to investigate the use of these specifications as the basis for a software tool which could help designers apply GoF patterns both consistently and correctly.

References

[1] A. Cechich and R. Moore, "A Formal Specification of GoF Design Patterns", Technical Report 151,

UNU/IIST, P.O.Box 3058, Macau, January, 1999.

[2] S. A. Cechich and R. Moore. "A Formal Specification of GoF Design Patterns". In *Proceeding of the Asia Pacific Software Engineering Conference: ASPEC'99*, Takamatsu, Japan, 1999, pp. 284-291.

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

[4] A. Eden, Y. Hirshfeld and A. Yehudai, "LePUS - A Declarative Pattern Specification Language". http://www.math.tau.ac.il/~eden/bibliography

[5] A. Eden, J.Gil and A. Yehudai, "A Formal Language for Design Patterns", The Department of Computer Science, School of Mathematics, Tel Aviv University of Israel, http://www.math.tau.ac.il/~eden/bibliography.html

[6] A. Flores and R. Moore, "GoF Structural Patterns: A Formal Specification". Technical Report 207, UNU/IIST, P.O.Box 3058, Macau, August, 2000. In *Proceeding of the IASTED International Conference on Applied Informatics (AI 2001)* , Innsbruck, Austria, 19-22 February 2001, pp. 625-630.

[7] A. Flores, L. Reynoso and R. Moore, "A Formal Model of Object-Oriented Design and GoF Design Patterns", Technical Report 200, UNU/IIST", P.O.Box 3058, Macau, August, 2000.

In *Proceeding of FME 2001: Formal Methods for Increasing Software Productivity*, Berlin, Germany, 12-16 March 2001, pp. 223-241.

[8] E. Gamma, R. Helm,R. Johnson and J. Vlissides. *"Design Patterns: Elements of Reusable Object-Oriented Software"*, Addison Wesley, 1995.

[9] L. Lamport, "The Temporal logics of Actions", *ACM Transactions on Programming Languages and Systems*, 16(3): 872-923, May 1924.

[10] T. Mikkonen, "Formalizing Design Patterns" , In *Proceeding of the International Conference on Software Engineering ICSE'98*, pp. 115-124.

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

[12] L. Reynoso and R. Moore. "GoF Behavioural Patterns: A Formal Specification", Technical Report 201, UNU/IIST, P.O.Box 3058, Macau, May, 2000.

[13] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen, *"Object-Oriented Modelling and Design"*, Prentice-Hall, 1991.

[14] L. M. Seiter, J. Palsberg and K. J. Lieberherr, "Evolution of Object Behavior using Context Relations", In *SIGSOFT 96*, pp. 46-57. ACM, 1996.